Handbuch zur dynamischen Ablaufverfolgung in Solaris

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Teilenr.: 819–6956–10
Oktober 2008

081119@21288

Inhalt

Abbildungen

Tabellen

Beispiele

Vorwort

DTrace ist ein umfassendes Framework für die dynamische Ablaufverfolgung im Betriebssystem Solaris™. DTrace bietet eine leistungsfähige Infrastruktur, die es Administratoren, Entwicklern und Wartungspersonal ermöglicht, beliebige Fragen zum Verhalten des Betriebssystems und der Benutzerprogramme zu beantworten. Im *Handbuch zur dynamischen Ablaufverfolgung in Solaris* wird beschrieben, wie sich das Systemverhalten mithilfe von DTrace beobachten, debuggen und abstimmen lässt. Darüber hinaus enthält dieses Buch einen vollständigen Referenzteil zu den in DTrace integrierten Beobachtungstools und zur Programmiersprache D.

Hinweis – Dieses Solaris-Release unterstützt Systeme auf der Basis der Prozessorarchitekturen SPARC® und x86: UltraSPARC®, SPARC64, AMD64, Pentium und Xeon EM64T. Die unterstützten Systeme können Sie in der *Solaris 10 Hardware-Kompatibilitätsliste* unter http://www.sun.com/bigadmin/hcl nachlesen. Eventuelle Implementierungsunterschiede zwischen den Plattformtypen sind in diesem Dokument angegeben.

Die Bezeichnung „x86" in diesem Dokument bezieht sich auf 64-Bit- und 32-Bit-Systeme mit AMD64- oder Intel Xeon-/Pentium-kompatiblen Prozessoren. Informationen zu unterstützten Systemen finden Sie in der *Solaris 10 Hardware-Kompatibilitätsliste*.

Zielgruppe dieses Handbuchs

Wenn Sie schon immer einmal das Verhalten Ihres Systems verstehen wollten, ist DTrace das richtige Tool für Sie. DTrace ist eine in Solaris integrierte, umfassende Einrichtung für die dynamische Ablaufverfolgung. DTrace kann sowohl zur Untersuchung des Verhaltens von Benutzerprogrammen als auch des Betriebssystemverhaltens eingesetzt werden. Das Framework ist zur Verwendung durch Systemadministratoren oder Anwendungsentwickler bestimmt und eignet sich für den Einsatz auf laufenden Produktionssystemen. DTrace bietet Ihnen die Möglichkeit, Ihr System genau zu durchleuchten, um seine Funktionsweise zu verstehen, Leistungsprobleme auf den verschiedensten Softwareebenen aufzuspüren oder die Ursachen von Fehlverhalten zu ermitteln. Wie Sie sehen werden, können Sie mit DTrace Ihre eigenen Programme für die dynamische Instrumentation des Systems und zur Ausgabe sofortiger, prägnanter Antworten auf beliebige Fragen schreiben, die Sie in der DTrace-Programmiersprache D formulieren.

Alle Solaris-Benutzer können mit DTrace:

- Tausende von Prüfpunkten dynamisch aktivieren und verwalten
- Prüfpunkten dynamisch logische Prädikate und Aktionen zuweisen
- Ablaufverfolgungspuffer und Pufferrichtlinien dynamisch verwalten
- Ablaufverfolgungsdaten aus einem laufenden System oder einem Speicherabzug anzeigen und untersuchen

Solaris-Entwickler und Administratoren können mit DTrace:

- benutzerdefinierte, das Dienstprogramm DTrace ansprechende Skripten implementieren
- mehrschichtige Tools implementieren, die DTrace zum Abrufen von Ablaufverfolgungsdaten verwenden

In diesem Handbuch erfahren Sie alles, was Sie über die Arbeit mit DTrace wissen müssen. Wenn Sie über Grundkenntnisse in einer Programmiersprache wie C oder einer Skriptsprache wie `awk(1)` oder `perl(1)` verfügen, können Sie sich DTrace und die Programmiersprache D zwar sicherlich schneller aneignen, Sie müssen jedoch kein Experte auf einem dieser Gebiete sein. Unter „Weiterführende Informationen“ auf Seite 24 sind Dokumente aufgeführt, die sich für Sie als hilfreich erweisen können, wenn Sie noch nie ein Programm oder Skript geschrieben haben.

Aufbau dieses Handbuchs

Kapitel 1, „Einführung“ stellt eine Blitztour durch das gesamte DTrace-Framework und eine Einführung in die Programmiersprache D dar. Kapitel 2, „Typen, Operatoren und Ausdrücke“, Kapitel 3, „Variablen“ und Kapitel 4, „D-Programmstruktur“ behandeln dann die Grundlagen von D ausführlicher und erläutern, wie D-Programme in dynamische Instrumentierung umgewandelt werden können. Es empfiehlt sich für alle Leser, diese erste Kapitelgruppe durchzulesen.

Kapitel 5, „Zeiger und Vektoren“, Kapitel 6, „Zeichenketten“, Kapitel 7, „Strukturen und Unionen“ und Kapitel 8, „Typ- und Konstantendefinitionen“ behandeln die übrigen Sprachmerkmale von D, von denen die meisten C-, C++- und Java™-Programmierern vertraut sein dürften. Leser, die mit keiner dieser Sprachen vertraut sind, sollten diese Kapitel lesen. Erfahrenere Programmierer können direkt mit den späteren Kapiteln fortfahren.

In Kapitel 9, „Aggregate“ und Kapitel 10, „Aktionen und Subroutinen“ werden das leistungsfähige DTrace-Grundelement zum Zusammenfassen von Daten in *Aggregaten* und der Satz integrierter Aktionen besprochen, die Ihnen beim Erstellen von Ablaufverfolgungsversuchen zur Verfügung stehen. Diese Kapitel sollten alle Leser aufmerksam durchlesen.

In Kapitel 11, „Puffer und Pufferung“ werden die DTrace-Richtlinien zum Puffern von Daten und ihre Konfigurationsmöglichkeiten erläutert. Lesen Sie dieses Kapitel, nachdem Sie sich mit dem Schreiben und Ausführen von D-Programmen vertraut gemacht haben.

In Kapitel 12, „Formatierung der Ausgabe“ werden die Aktionen für die Ausgabeformatierung in D sowie die Standardrichtlinie für die Formatierung der Ablaufverfolgungsdaten beschrieben. Wer mit der C-Funktion `printf()` vertraut ist, braucht dieses Kapitel nur flüchtig zu lesen. Leser, die `printf()` noch nie begegnet sind, sollten das Kapitel aufmerksam durchlesen.

Kapitel 13, „Spekulative Ablaufverfolgung“ beleuchtet DTrace im Hinblick auf die *spekulative* Übergabe von Daten an einen Ablaufverfolgungspuffer. Dieses Kapitel richtet sich an Benutzer, die DTrace zum Verfolgen von Daten einsetzen müssen, noch bevor deren Relevanz für die bestehende Fragestellung geklärt ist.

Kapitel 14, „Das Dienstprogramm `dtrace`(1M)“ stellt, vergleichbar mit der entsprechenden Seite im Online-Handbuch, eine vollständige Referenz für das Befehlszeilendienstprogramm `dtrace` dar. Hier können Leser nachschlagen, wenn an anderer Stelle im Buch unbekannte Befehlszeilenoptionen erwähnt werden. In Kapitel 15, „Scripting“ wird erläutert, wie sich mit dem Dienstprogramm `dtrace` ausführbare D-Skripten konstruieren und deren Befehlszeilenargumente verarbeiten lassen. In Kapitel 16, „Optionen und Tunables “ werden die Optionen beschrieben, die in der Befehlszeile oder innerhalb eines D-Programms selbst angepasst werden können.

Die Gruppe der Kapitel ab Kapitel 17, „Der Provider `dtrace`“ bis einschließlich Kapitel 32, „Der Provider `fasttrap`“ behandelt die DTrace-*Provider*, die zum Instrumentieren verschiedener Aspekte des Solaris-Systems dienen. Alle Leser sollten diese Kapitel durchblättern, um sich mit den verschiedenen Providern vertraut zu machen, und anschließend die genaue Lektüre der jeweils benötigten Kapitel fortsetzen.

Kapitel 33, „Ablaufverfolgung von Benutzerprozessen“ enthält Beispiele zum Instrumentieren von Benutzerprozessen mit DTrace. Kapitel 34, „Statisch definierte Ablaufverfolgung für Benutzeranwendungen“ erläutert, wie Anwendungsprogrammierer benutzerspezifische DTrace-Provider und -Prüfpunkte in Benutzeranwendungen einfügen können. Entwicklern oder Administratoren von Benutzerprogrammen, die mit DTrace das Verhalten von Benutzerprozessen untersuchen möchten, wird die Lektüre dieser Kapitel empfohlen.

Kapitel 35, „Sicherheit“ und die übrigen Kapitel befassen sich mit fortgeschrittenen Themen wie Sicherheits-, Versions- und Stabilitätsattributen von DTrace sowie der Durchführung von Ablaufverfolgungen mit DTrace beim Booten und nach Systemabstürzen. Diese Kapitel richten sich an erfahrene DTrace-Benutzer.

Weiterführende Informationen

Die folgenden Bücher und Schriften, deren Lektüre empfohlen wird, beziehen sich auf Vorgänge, die Sie bei der Arbeit mit DTrace durchführen müssen:

- Kernighan, Brian W. und·Ritchie, Dennis M. *The C Programming Language*. Prentice Hall, 1988. ISBN 0–13–110370–9
- Vahalia, Uresh. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996. ISBN 0-13-101908-2
- Mauro, Jim and McDougall, Richard. *Solaris Internals: Core Kernel Components*. Sun Microsystems Press, 2001. ISBN 0-13-022496-0

Unter http://www.sun.com/bigadmin/content/dtrace/ können Sie Ihre Erfahrungen und DTrace-Skripten mit dem Rest der DTrace-Community teilen.

Dokumentation, Support und Schulung

Auf der Sun-Website finden Sie Informationen zu den folgenden zusätzlichen Ressourcen:

- Dokumentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Schulung (`http://www.sun.com/training/`)

Typografische Konventionen

In der folgenden Tabelle sind die in diesem Handbuch verwendeten typografischen Konventionen aufgeführt.

TABELLE P–1 Typografische Konventionen

Schriftart	Bedeutung	Beispiel
`AaBbCc123`	Die Namen von Befehlen, Dateien, Verzeichnissen sowie Bildschirmausgabe.	Bearbeiten Sie Ihre `.login`-Datei. Verwenden Sie `ls -a`, um eine Liste aller Dateien zu erhalten. `system% Sie haben eine neue Nachricht.`
`AaBbCc123`	Von Ihnen eingegebene Zeichen (im Gegensatz zu auf dem Bildschirm angezeigten Zeichen)	`system%` **`su`** `Passwort:`

TABELLE P–1 Typografische Konventionen *(Fortsetzung)*

Schriftart	Bedeutung	Beispiel
aabbcc123	Platzhalter: durch einen tatsächlichen Namen oder Wert zu ersetzen	Geben Sie zum Löschen einer Datei den Befehl `rm` *Dateiname* ein.
AaBbCc123	Buchtitel, neue Ausdrücke; hervorgehobene Begriffe	Lesen Sie hierzu Kapitel 6 im *Benutzerhandbuch*.
		Ein *Cache* ist eine lokal gespeicherte Kopie.
		Diese Datei *nicht* speichern.
		Hinweis: Einige hervorgehobene Begriffe werden online fett dargestellt.

Shell-Eingabeaufforderungen in Befehlsbeispielen

Die folgende Tabelle zeigt die Standard-Systemeingabeaufforderung von UNIX® und die Superuser-Eingabeaufforderung für die C-Shell, die Bourne-Shell und die Korn-Shell.

TABELLE P–2 Shell-Eingabeaufforderungen

Shell	Eingabeaufforderung
C-Shell	`system%`
C-Shell für Superuser	`system#`
Bourne-Shell und Korn-Shell	`$`
Bourne-Shell und Korn-Shell für Superuser	`#`

◆ ◆ ◆ KAPITEL 1

1 Einführung

Willkommen bei der dynamischen Ablaufverfolgung im Betriebssystem Solaris! Wenn Sie schon immer einmal das Verhalten Ihres Systems verstehen wollten, ist DTrace das richtige Tool für Sie. DTrace ist ein in Solaris integriertes umfassendes Framework für die dynamische Ablaufverfolgung (auch „Tracing“ genannt), das sowohl von Administratoren als auch Entwicklern zur Untersuchung des Verhaltens von Benutzerprogrammen und des Betriebssystems auf laufenden Produktionssystemen eingesetzt werden kann. DTrace bietet Ihnen die Möglichkeit, Ihr System genau zu durchleuchten, um seine Funktionsweise zu verstehen, Leistungsprobleme auf den verschiedensten Softwareebenen aufzuspüren oder die Ursachen von Fehlverhalten zu ermitteln. Wie Sie sehen werden, können Sie mit DTrace Ihre eigenen Programme für die dynamische Instrumentation des Systems und zur Ausgabe sofortiger, prägnanter Antworten auf beliebige Fragen schreiben, die Sie in der DTrace-Programmiersprache D formulieren. Im ersten Teil dieses Kapitels erhalten Sie eine schnelle Einführung in DTrace und erfahren, wie Sie Ihr erstes D-Programm schreiben können. Im verbleibenden Teil des Kapitels werden der vollständige Regelsatz für die Programmierung in D vorgestellt und Tipps und Techniken für tief greifende Analysen des Systems erläutert. Unter http://www.sun.com/bigadmin/content/dtrace/ können Sie Ihre Erfahrungen und DTrace-Skripten mit dem Rest der DTrace-Gemeinde teilen. Alle in diesem Handbuch aufgeführten Beispielskripten finden Sie auf Ihrem Solaris-System im Verzeichnis `/usr/demo/dtrace`.

Erste Schritte

DTrace hilft Ihnen, Softwaresysteme genau zu verstehen, indem Sie den Betriebssystemkernel und Benutzerprozesse dynamisch so verändern, dass an Stellen besonderen Interesses, den so genannten *Prüfpunkten*, zusätzliche, von Ihnen angegebene Informationen aufgezeichnet werden. Ein Prüfpunkt ist eine Stelle oder eine Aktivität, an die DTrace eine Anforderung zur Durchführung einer Gruppe von *Aktionen* binden kann. Dabei kann es sich beispielsweise um die Aufzeichnung eines Stackprotokolls, einer Zeitmarke oder des Arguments einer Funktion handeln. Prüfpunkte sind mit programmierbaren Sensoren oder Messfühlern vergleichbar, die an interessanten Stellen im gesamten Solaris-System zu finden sind. Wenn Sie einer Sache auf

den Grund gehen möchten, programmieren Sie mit DTrace die entsprechenden Sensoren so, dass die für Sie interessanten Informationen aufgezeichnet werden. Wenn anschließend die Prüfpunkte *ausgelöst werden*, werden die Daten der Prüfpunkte von DTrace abgerufen und an Sie ausgegeben. Wenn Sie für einen Prüfpunkt keine Aktion festlegen, verzeichnet DTrace lediglich, wie oft der Prüfpunkt ausgelöst wird.

Jeder Prüfpunkt in DTrace besitzt zwei Namen: eine eindeutige, ganzzahlige ID und einen vom Menschen lesbaren textuellen Namen. Zum Erlernen von DTrace sollen zunächst einige sehr einfache Anforderungen mit dem Prüfpunkt namens `BEGIN` erzeugt werden, der zu Beginn jeder neuen Tracing-Anforderung einmal ausgelöst wird. Mit der Option des Dienstprogramms`dtrace(1M)` lässt `-n` sich ein Prüfpunkt über seinen Zeichenfolgennamen aktivieren. Geben Sie den folgenden Befehl ein:

```
# dtrace -n BEGIN
```

Nach einer kurzen Pause wird Ihnen von DTrace mitgeteilt, dass ein Prüfpunkt aktiviert wurde, und Sie sehen eine Ausgabezeile, die auf die Auslösung des Prüfpunkts `BEGIN` hinweist. Wenn diese Ausgabe erscheint, wartet `dtrace` darauf, dass weitere Prüfpunkte ausgelöst werden. Da Sie keine weiteren Prüfpunkte aktiviert haben und `BEGIN` nur einmal ausgelöst wird, drücken Sie in Ihrer Befehls-Shell die Tastenkombination Strg-C, um `dtrace` zu beenden und zur Shell-Eingabeaufforderung zurückzukehren:

```
# dtrace -n BEGIN
dtrace: description 'BEGIN' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                      :BEGIN
^C
#
```

An der Ausgabe erkennen Sie, dass der Prüfpunkt namens `BEGIN` einmal ausgelöst wurde. Sie enthält sowohl die ganzzahlige ID als auch den Namen des Prüfpunkts. Standardmäßig wird die numerische Bezeichnung der CPU angezeigt, auf der der Prüfpunkt ausgelöst wird. In diesem Beispiel geht aus der Spalte „CPU“ hervor, dass der Befehl `dtrace` bei Auslösung des Prüfpunkts auf CPU 0 ausgeführt wurde.

Sie können DTrace-Anforderungen mit beliebig vielen Prüfpunkten und Aktionen erstellen. Formulieren Sie nun eine einfache Anforderung mit zwei Prüfpunkten, indem Sie dem vorigen Beispielbefehl den Prüfpunkt `END` hinzufügen. Der Prüfpunkt `END` wird einmal zum Abschluss der Ablaufverfolgung ausgelöst. Geben Sie den folgenden Befehl ein und drücken Sie nach der Ausgabezeile für den Prüfpunkt `BEGIN` erneut Strg-C in Ihrer Befehls-Shell:

```
# dtrace -n BEGIN -n END
dtrace: description 'BEGIN' matched 1 probe
dtrace: description 'END' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                      :BEGIN
```

```
^C
  0      2                     :END
#
```

Sie sehen, dass die Betätigung von Strg-C zum Beenden von `dtrace` den Prüfpunkt `END` auslöst. `dtrace` meldet die Auslösung dieses Prüfpunkts und wird beendet.

Nachdem Sie sich jetzt ein wenig mit der Benennung und Aktivierung von Prüfpunkten angefreundet haben, sind Sie bereit, die DTrace-Version des klassischen Einsteigerprogramms „Hello, World“ zu schreiben. DTrace-Versuche lassen sich nicht nur in der Befehlszeile erstellen, sondern auch in der Programmiersprache D in Textdateien schreiben. Erstellen Sie in einem Texteditor eine neue Datei mit dem Namen `hello.d` und geben Sie Ihr erstes D-Programm ein:

BEISPIEL 1–1 `hello.d`: „Hello, World“ in der Programmiersprache D

```
BEGIN
{
    trace("hello, world");
    exit(0);
}
```

Nachdem Sie das Programm gespeichert haben, können Sie es mit der `dtrace`-Option `-s` ausführen. Geben Sie den folgenden Befehl ein:

```
# dtrace -s hello.d
dtrace: script 'hello.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                      :BEGIN   hello, world
#
```

Wie Sie sehen, generiert `dtrace` dieselbe Ausgabe wie zuvor, gefolgt von dem Text „`hello, world`”. Im Gegensatz zum vorherigen Beispiel mussten Sie diesmal weder warten noch Strg-C drücken. Diese Abweichungen sind das Ergebnis der *Aktionen*, die Sie in `hello.d` für den Prüfpunkt `BEGIN` angegeben haben. Um verstehen zu können, was hier vor sich gegangen ist, müssen wir uns die Struktur des D-Programms genauer anschauen.

Jedes D-Programm besteht aus einer Folge von *Klauseln*, die je einen oder mehrere zu aktivierende Prüfpunkte beschreiben, und einem optionalen Satz Aktionen, die bei Auslösung des Prüfpunkts durchgeführt werden sollen. Die Aktionen werden nach dem Namen des Prüfpunkts als eine Reihe von Anweisungen in geschweiften Klammern `{ }` aufgelistet. Jede Anweisung endet mit einem Strichpunkt (`;`). In Ihrer ersten Anweisung kommt die Funktion `trace()` zum Einsatz. Mit ihr weisen Sie DTrace an, das angegebene Argument, die Zeichenkette „`hello, world`”, bei Auslösung des Prüfpunkts `BEGIN` aufzuzeichnen und anschließend auszugeben. In der zweiten Anweisung verwenden Sie die Funktion `exit()`, um die Ablaufverfolgung mit DTrace abzubrechen und den Befehl `dtrace` zu beenden. DTrace

bietet einen Satz hilfreicher Funktionen wie `trace()` und `exit()`, die Sie in Ihren D-Programmen aufrufen können. Sie rufen eine Funktion auf, indem Sie ihren Namen, gefolgt von einer in Klammern stehenden Liste von Argumenten angeben. Alle D-Funktionen werden in Kapitel 10, „Aktionen und Subroutinen“ beschrieben.

Wenn Sie mit der Programmiersprache C vertraut sind, haben Sie an dem Namen und unseren Beispielen mittlerweile sicherlich festgestellt, dass sie der DTrace-Programmiersprache D sehr ähnlich ist. Tatsächlich ist D eine Ableitung eines sehr weiten Teilsatzes von C, kombiniert mit einem speziellen, die Ablaufverfolgung erleichternden Funktions- und Variablensatz. In den nachfolgenden Kapiteln werden Sie mehr über diese Leistungsmerkmale erfahren. Wenn Sie schon einmal ein C-Programm geschrieben haben, können Sie Ihr Wissen großteilig unmittelbar auf die Erstellung von Ablaufverfolgungsprogrammen in D übertragen. Aber selbst wenn Sie noch nie ein C-Programm geschrieben haben, lässt sich D sehr leicht erlernen. In diesem Kapitel wird Ihnen die gesamte Syntax verständlich gemacht. Lassen Sie uns aber zunächst noch einmal von den Sprachregeln zur Funktionsweise von DTrace und anschließend zum Erstellen etwas interessanterer D-Programme zurückkehren.

Provider und Prüfpunkte

In den vorangehenden Beispielen haben Sie die Verwendung zwei einfacher Prüfpunkte namens `BEGIN` und `END` erlernt. Woher kamen diese Prüfpunkte eigentlich? Prüfpunkte in DTrace stammen aus einer Gruppe von Kernelmodulen, den *Providern*. Jeder Provider nimmt eine bestimmte Art der Instrumentation zur Erstellung von Prüfpunkten vor. Bei der Verwendung von DTrace erhält jeder Provider die Gelegenheit, die Prüfpunkte zu veröffentlichen, die er dem DTrace-Framework zur Verfügung stellt. Sie können dann Ablaufverfolgungsaktionen aktivieren und einem beliebigen der veröffentlichten Prüfpunkte zuordnen. Zum Auflisten aller auf dem System verfügbaren Prüfpunkte geben Sie folgenden Befehl ein:

```
# dtrace -l
  ID   PROVIDER            MODULE          FUNCTION NAME
   1     dtrace                                     BEGIN
   2     dtrace                                     END
   3     dtrace                                     ERROR
   4   lockstat          genunix       mutex_enter adaptive-acquire
   5   lockstat          genunix       mutex_enter adaptive-block
   6   lockstat          genunix       mutex_enter adaptive-spin
   7   lockstat          genunix       mutex_exit  adaptive-release

   ... many lines of output omitted ...

#
```

Die Anzeige der gesamten Ausgabe kann eine Weile dauern. Für die Anzeige der Gesamtanzahl aller Prüfpunkte geben Sie folgenden Befehl ein:

```
# dtrace -l | wc -l
        30122
```

Die Anzahl auf Ihrem System kann hiervon abweichen, da sie von der jeweiligen Betriebsplattform und der installierten Software abhängt. Die enorme Menge an verfügbaren Prüfpunkten gewährt Ihnen einen Einblick in jeden bislang dunklen Winkel des Systems. Selbst diese Ausgabe ist noch keine vollständige Liste aller Prüfpunkte, da einige Provider, wie Sie später erfahren werden, die Möglichkeit bieten, auf Grundlage Ihrer Tracing-Anforderungen neue Prüfpunkte on-the-fly zu erstellen und die tatsächliche Anzahl der DTrace-Prüfpunkte somit schier ins Unendliche treiben.

Schauen Sie sich noch einmal die Ausgabe von **`dtrace -l`** im Terminalfenster an. Beachten Sie, dass jeder Prüfpunkt mit den zwei zuvor genannten Namen, der ganzzahligen ID und dem textuellen Namen, aufgeführt ist. Der textuelle Name besteht aus vier Teilen, die in der `dtrace`-Ausgabe als vier separate Spalten angezeigt werden. Es werden die folgenden Bestandteile eines Prüfpunktnamens unterschieden:

Provider	Der Name des DTrace-Providers, der diesen Prüfpunkt veröffentlicht. Der Name des Providers stimmt bezeichnenderweise mit dem Namen des DTrace-Kernelmoduls überein, das die Instrumentation zur Aktivierung des Prüfpunkts durchführt.
Modul	Bei Prüfpunkten für eine bestimmte Programmposition der Name des Moduls, in dem sich der Prüfpunkt befindet. Dabei handelt es sich entweder um den Namen eines Kernelmoduls oder einer Benutzerbibliothek.
Funktion	Bei Prüfpunkten für eine bestimmte Programmposition der Name der Programmfunktion, in der sich der Prüfpunkt befindet.
Name	Der letzte Bestandteil des Prüfpunktnamens gibt, wie beispielsweise `BEGIN` oder `END`, in gewissem Maße Aufschluss über die semantische Bedeutung des Prüfpunkts.

Geben Sie beim Ausschreiben des vollständigen textuellen Namens eines Prüfpunkts alle vier, durch Doppelpunkt getrennte Bestandteile an:

Provider:*Modul*: *Funktion*:*Name*

Für einige Prüfpunkte, wie beispielsweise `BEGIN` und `END`, sind in der Liste weder Modul noch Funktion aufgeführt. Bei einigen Prüfpunkten bleiben diese Felder leer, da sie keiner bestimmten instrumentierten Programmfunktion oder -position entsprechen. Sie beziehen sich vielmehr auf ein abstraktes Konzept wie zum Beispiel das Ende einer Tracing-Anforderung. Prüfpunkte, deren Namen Modul- und Funktionsbestandteile enthalten, werden als *verankerte Prüfpunkte* bezeichnet, solche ohne diese Bestandteile als *nicht verankerte Prüfpunkte*.

Wenn Sie nicht alle Felder eines Prüfpunktnamens angeben, wird Ihre Anforderung von DTrace konventionsgemäß auf *alle* Prüfpunkte mit übereinstimmenden Werten in den von Ihnen angegebenen Namensbestandteilen angewendet. Das heißt also, dass Sie DTrace zuvor

mit der Angabe des Prüfpunktnamens `BEGIN` angewiesen haben, unabhängig von den Werten im Provider-, Modul- und Funktionsfeld alle Prüfpunkte mit dem Namensfeld `BEGIN` zu suchen. Zufällig gibt es nur einen Prüfpunkt mit dieser Beschreibung, und deshalb fällt das Ergebnis gleich aus. Doch Sie wissen nun, dass `dtrace:::BEGIN` der richtige Name des Prüfpunkts `BEGIN` ist und darauf hinweist, dass dieser Prüfpunkt vom DTrace-Framework selbst bereitgestellt wird (DTrace ist sein Provider) und an keiner Funktion verankert ist. Wir könnten das Programm `hello.d` also auch wie folgt schreiben und dasselbe Ergebnis erhalten:

```
dtrace:::BEGIN
{
    trace("hello, world");
    exit(0);
}
```

Nachdem Sie jetzt wissen, woher Prüfpunkte kommen und wie sie benannt werden, betrachten wir ein wenig genauer, was geschieht, wenn Sie Prüfpunkte aktivieren und DTrace zu einem Vorgang anweisen. Anschließend kehren wir zu unserem Blitzkurs in D zurück.

Kompilierung und Instrumentation

Beim Schreiben herkömmlicher Programme für Solaris konvertieren Sie den Quellcode des Programms mit einem Compiler in den ausführbaren Objektcode. Indem Sie den Befehl `dtrace` absetzen, rufen Sie den Compiler für die Programmiersprache D auf, in der Sie zuvor das Programm `hello.d` geschrieben haben. Das kompilierte Programm wird an den Betriebssystemkernel gesendet, wo es durch DTrace ausgeführt wird. Dort werden die im Programm genannten Prüfpunkte aktiviert und der entsprechende Provider nimmt die erforderliche Instrumentation vor, um sie in Funktion zu setzen.

Die gesamte Instrumentation in DTrace erfolgt vollständig dynamisch: Prüfpunkte werden einzeln und nur, wenn Sie sie verwenden, aktiviert. Für inaktive Prüfpunkte liegt kein instrumentierter Code vor, sodass das System keinerlei Leistungseinbußen erfährt, wenn DTrace nicht verwendet wird. Wenn das Experiment abgeschlossen und der Befehl `dtrace` beendet wird, werden alle von Ihnen verwendeten Prüfpunkte automatisch deaktiviert und ihre Instrumentation gelöscht. Dadurch kehrt das System zu exakt dem vorherigen Zustand zurück. Zwischen einem System, auf dem DTrace nicht aktiv ist, und einem System, auf dem die DTrace-Software nicht installiert ist, besteht effektiv kein Unterschied.

Die Instrumentation für jeden Prüfpunkt erfolgt dynamisch auf dem laufenden Betriebssystem oder den von Ihnen ausgewählten Benutzerprozessen. Das System wird weder stillgelegt (quiesce) noch auf eine andere Weise unterbrochen. Nur für die von Ihnen aktivierten Prüfpunkte wird Instrumentationscode hinzugefügt. Das heißt, dass die Prüftätigkeit durch die Verwendung von DTrace genau darauf beschränkt ist, was Sie von DTrace verlangen: Keine anderen Daten werden nachverfolgt, kein einzelner, großer „Tracing-Schalter“ wird im System

eingeschaltet. Die gesamte Instrumentation in DTrace ist auf maximale Effizienz ausgelegt. Dank dieser Leistungsmerkmale lässt sich DTrace in der Produktionsumgebung zum Lösen echter Probleme in Echtzeit einsetzen.

Das DTrace-Framework bietet außerdem Unterstützung für eine beliebige Anzahl virtueller Clients. Sie können so viele DTrace-Experimente gleichzeitig ausführen, wie Sie für nötig halten. Dabei stellt die Speicherkapazität der Systeme die einzige Einschränkung dar. Alle Befehle arbeiten unabhängig voneinander unter Verwendung derselben zugrunde liegenden Instrumentation. Derselben Fähigkeit ist es auch zu verdanken, dass beliebig viele unterschiedliche Systembenutzer DTrace gleichzeitig einsetzen können: Entwickler, Administratoren und Wartungsfachleute können mit DTrace gemeinsam oder an getrennten Problemen auf demselben System arbeiten, ohne sich gegenseitig zu behindern.

Im Gegensatz zu Programmen in C und C++ und ähnlich wie in der Programmiersprache Java™ geschriebene Programme werden die D-Programme in DTrace in eine sichere Zwischenform kompiliert, die bei der Auslösung der Prüfpunkte ausgeführt wird. Die Sicherheitsüberprüfung dieser Zwischenform erfolgt, wenn das Programm erstmals von der DTrace-Kernelsoftware untersucht wird. Die DTrace-Ausführungsumgebung behandelt außerdem etwaige während der Ausführung des D-Programms auftretende Laufzeitfehler, einschließlich Divisionen durch Null, Dereferenzierungen ungültiger Speicherbereiche usw., und gibt Meldungen darüber aus. Sie können also gar kein unsicheres Programm erzeugen, das DTrace dazu veranlassen würde, den Solaris-Kernel oder einen der auf dem System ausgeführten Prozesse zu beschädigen. Aufgrund dieser Sicherheitsmerkmale können Sie DTrace in Produktionsumgebungen einsetzen, ohne sich über Abstürze oder Beschädigungen des Systems Gedanken machen zu müssen. Im Fall eines Programmierfehlers meldet DTrace den Fehler und deaktiviert die Instrumentation, sodass Sie den Fehler korrigieren und einen neuen Versuch starten können. Die DTrace-Leistungsmerkmale zum Melden von Fehlern und Debuggen werden später in diesem Buch beschrieben.

Das folgende Schaubild zeigt die einzelnen Komponenten der DTrace-Architektur, einschließlich der Provider, Prüfpunkte, DTrace-Kernelsoftware und des Befehls `dtrace`.

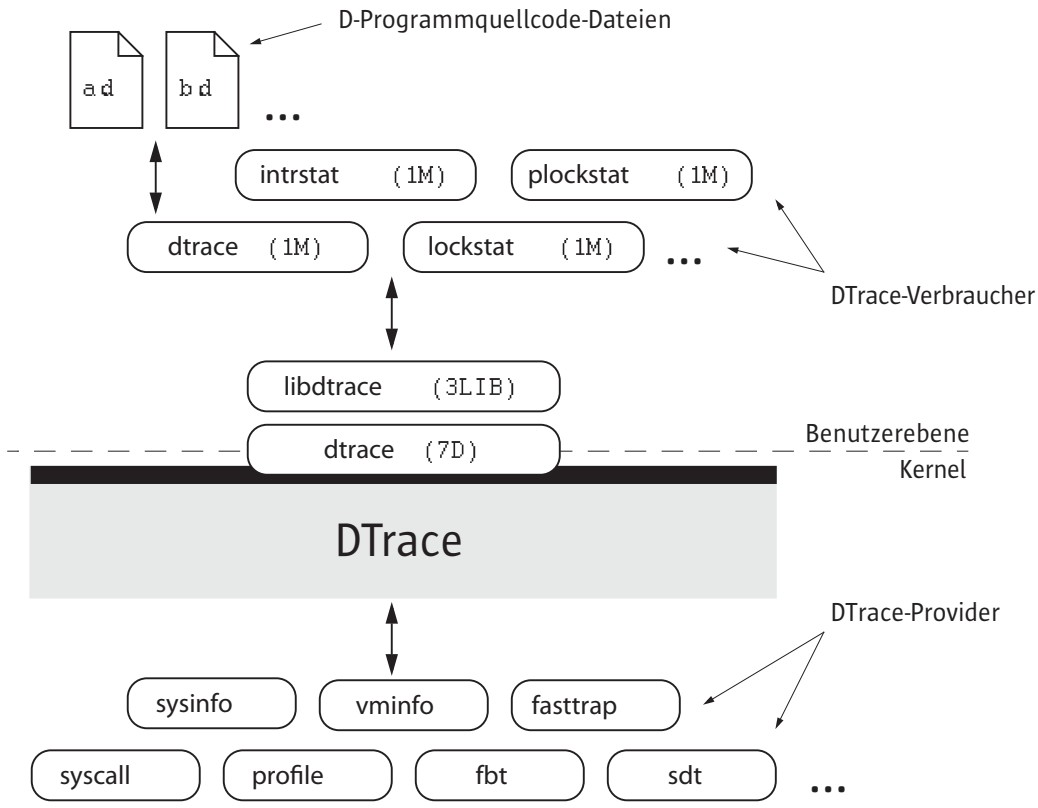

ABBILDUNG 1–1 DTrace-Architektur und -Komponenten im Überblick

Nun, nachdem die Funktionsweise von DTrace geklärt ist, kehren wir zur Einführung in die Programmiersprache D zurück und beginnen, einige interessantere Programme zu schreiben.

Variablen und arithmetische Ausdrücke

In unserem nächsten Beispielprogramm wird mit dem DTrace-Provider `profile` ein einfacher, auf der verstreichenden Zeit basierender Zähler implementiert. Dieser Provider hat die Fähigkeit, neue Prüfpunkte auf der Grundlage der im D-Programm enthaltenen Beschreibungen zu erstellen. Wenn Sie einen Prüfpunkt namens `profile:::tick-`*n*`sec` für die Ganzzahl *n* erstellen, generiert der Provider einen Prüfpunkt, der alle *n* Sekunden ausgelöst wird. Schreiben Sie den folgenden Quellcode und speichern Sie ihn in einer Datei namens `counter.d`:

```
/*
 * Count off and report the number of seconds elapsed
 */
```

```
dtrace:::BEGIN
{
    i = 0;
}

profile:::tick-1sec
{
    i = i + 1;
    trace(i);
}

dtrace:::END
{
    trace(i);
}
```

Wenn das Programm ausgeführt wird, werden die ablaufenden Sekunden gezählt, bis Sie Strg-C drücken, und anschließend wird die Gesamtzahl angezeigt:

```
# dtrace -s counter.d
dtrace: script 'counter.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
  0  25499                       :tick-1sec        1
  0  25499                       :tick-1sec        2
  0  25499                       :tick-1sec        3
  0  25499                       :tick-1sec        4
  0  25499                       :tick-1sec        5
  0  25499                       :tick-1sec        6
^C
  0      2                             :END        6
#
```

Die ersten drei Programmzeilen sind ein Kommentar, aus dem die Funktion des Programms hervorgeht. Ähnlich wie in C, C++ und der Programmiersprache Java ignoriert der D-Compiler alle Zeichen zwischen den Symbolen `/*` und `*/`. Kommentare sind an jeder Stelle in einem D-Programm zulässig, sowohl innerhalb als auch außerhalb der Prüfpunkt-Klausel.

Die Klausel für den Prüfpunkt `BEGIN` definiert eine neue Variable namens `i` und weist ihr mit folgender Anweisung den ganzzahligen Wert Null zu:

```
i = 0;
```

Im Gegensatz zu C, C++ und Java lassen sich D-Variablen einfach durch ihre Verwendung in einer Programmanweisung erzeugen. Es sind keine expliziten Variablendeklarationen erforderlich. Bei der ersten Verwendung einer Variablen in einem Programm wird der Variablentyp auf Grundlage der Art ihrer ersten Zuweisung festgelegt. Eine Variable kann im Verlauf der Programmlebensdauer nur einen Typ aufweisen. Nachfolgende Referenzen müssen also dem Typ der ersten Zuweisung entsprechen. In `counter.d` wird der Variable `i` zuerst die

ganzzahlige Konstante Null zugewiesen. Sie wird deshalb auf den Datentyp `int` festgelegt. D bietet dieselben einfachen Integer-Datentypen wie C:

`char`	Zeichen oder Ganzzahl von 1 Byte
`int`	Standard-Integer
`short`	Kurze Ganzzahl
`long`	Lange Ganzzahl
`long long`	Sehr lange Ganzzahl

Die Größen dieser Typen hängen von dem in Kapitel 2, „Typen, Operatoren und Ausdrücke“ beschriebenen Datenmodell des Betriebssystemkerns ab. Darüber hinaus bietet D integrierte einfache Namen für Integer-Typen mit und ohne Vorzeichen von unterschiedlicher, festgelegter Größe sowie Tausende anderer Typen, die vom Betriebssystem definiert werden.

Der zentrale Teil von `counter.d` ist die Prüfpunktklausel für die Erhöhung des Zählers `i`:

```
profile:::tick-1sec
{
    i = i + 1;
    trace(i);
}
```

Mit dieser Klausel wird der Prüfpunkt `profile:::tick-1sec` benannt. Dies weist den Provider `profile` an, einen neuen Prüfpunkt zu erzeugen, der einmal pro Sekunde auf einem verfügbaren Prozessor ausgelöst wird. Die Klausel enthält zwei Anweisungen. Die erste ordnet `i` den vorigen Wert plus 1 zu und die zweite überwacht den neuen Wert von `i`. In D sind alle aus C bekannten arithmetischen Operatoren verfügbar. Eine vollständige Liste finden Sie in Kapitel 2, „Typen, Operatoren und Ausdrücke“. Ebenso wie in C kann der Operator ++ als Kurzform für die Erhöhung der entsprechenden Variable um 1 verwendet werden. Die Funktion `trace()` übernimmt jeden D-Ausdruck als Argument. `counter.d` ließe sich also auch knapper fassen:

```
profile:::tick-1sec
{
    trace(++i);
}
```

Wenn Sie den Typ der Variable `i` ausdrücklich festlegen möchten, können Sie den gewünschten Typ bei der Zuweisung einklammern, um die Ganzzahl Null ausdrücklich in einen bestimmten Typ umzuwandeln (*Casting*). Wenn Sie in D beispielsweise die maximale Größe einer `char`-Variable ermitteln möchten, könnten Sie die `BEGIN`-Klausel wie folgt ändern:

```
dtrace:::BEGIN
{
    i = (char)0;
}
```

Wenn Sie `counter.d` eine Weile laufen lassen, müssten Sie sehen, wie der überwachte Wert anwächst und schließlich wieder bei Null beginnt. Sollte Ihnen das zu lange dauern, ändern Sie den Namen des Prüfpunkts `profile` in `profile:::tick-100msec` ab. Dadurch erhalten Sie einen Zähler, der alle 100 Millisekunden (10-mal pro Sekunde) erhöht wird.

Prädikate

Einer der Hauptunterschiede zwischen D und anderen Programmiersprachen wie C, C++ und Java besteht im Verzicht auf Kontrollstrukturen wie beispielsweise if-Anweisungen und Schleifen. Klauseln von D-Programmen werden als einfache, geradlinige Anweisungslisten geschrieben, die eine optionale, festgelegte Menge an Daten verfolgen. D bietet die Möglichkeit der bedingten Datenverfolgung und der Einflussnahme auf den Kontrollfluss anhand von logischen Ausdrücken namens *Prädikaten*, die Programmklauseln vorangesetzt werden können. Ein Prädikatausdruck wird bei der Prüfpunktauslösung ausgewertet, bevor die Anweisungen in der entsprechenden Klausel ausgeführt werden. Wenn die Prädikatauswertung „wahr“ ergibt, was durch einen beliebigen Wert ungleich Null dargestellt wird, erfolgt die Ausführung der Anweisungsliste. Ergibt die Prädikatauswertung bei einem Wert von Null „falsch“, wird keine der Anweisungen ausgeführt und die Auslösung des Prüfpunkts ignoriert.

Schreiben Sie für das nächste Beispiel den folgenden Quellcode und speichern Sie ihn in einer Datei namens `countdown.d`:

```
dtrace:::BEGIN
{
    i = 10;
}

profile:::tick-1sec
/i > 0/
{
    trace(i--);
}

profile:::tick-1sec
/i == 0/
{
    trace("blastoff!");
    exit(0);
}
```

Dieses D-Programm implementiert unter Verwendung von Prädikaten einen Timer für einen 10-sekündigen Countdown. Wenn `countdown.d` ausgeführt wird, zählt das Programm von 10 abwärts, gibt anschließend eine Meldung aus und wird beendet:

```
# dtrace -s countdown.d
dtrace: script 'countdown.d' matched 3 probes
CPU     ID                    FUNCTION:NAME
    0  25499                         :tick-1sec        10
    0  25499                         :tick-1sec         9
    0  25499                         :tick-1sec         8
    0  25499                         :tick-1sec         7
    0  25499                         :tick-1sec         6
    0  25499                         :tick-1sec         5
    0  25499                         :tick-1sec         4
    0  25499                         :tick-1sec         3
    0  25499                         :tick-1sec         2
    0  25499                         :tick-1sec         1
    0  25499                         :tick-1sec   blastoff!
#
```

In diesem Beispiel wird mit dem Prüfpunkt `BEGIN` zum Starten des Countdowns ein Integer `i` auf 10 initialisiert. Wie im vorigen Beispiel wird anschließend mit dem Prüfpunkt `tick-1sec` ein einmal pro Sekunde ausgelöster Timer implementiert. Beachten Sie, dass die Beschreibung des Prüfpunkts `tick-1sec` in `countdown.d` in zwei verschiedenen Klauseln und mit zwei verschiedenen Prädikaten und Aktionslisten verwendet wird. Das Prädikat ist ein zwischen Schrägstrichen `/ /` stehender logischer Ausdruck, der dem Prüfpunktnamen folgt und der geschweiften Klammer `{ }` um die Klausel-Anweisungsliste vorangeht.

Das erste Prädikat testet, ob `i` größer als Null ist, was darauf hinweist, dass der Timer noch läuft:

```
profile:::tick-1sec
/i > 0/
{
    trace(i--);
}
```

Der relationale Operator `>` bedeutet *größer als* und gibt als „falsch“ den ganzzahligen Wert Null und als „wahr“ den Wert 1 zurück. In D werden alle aus C bekannten relationalen Operatoren unterstützt. Eine vollständige Liste finden Sie in Kapitel 2, „Typen, Operatoren und Ausdrücke“. Wenn `i` noch nicht Null ist, wird `i` durch das Skript überwacht und dann mithilfe des Operators `- -` verringert.

Das zweite Prädikat verwendet den Operator `==`, um „wahr“ zurückzugeben, wenn `i` genau gleich Null ist. Dies weist darauf hin, dass der Countdown abgeschlossen ist:

```
profile:::tick-1sec
/i == 0/
{
```

```
    trace("blastoff!");
    exit(0);
}
```

Ähnlich wie in unserem ersten Beispiel `hello.d` kommt in `countdown.d` eine Folge von Zeichen in Anführungsstrichen zum Einsatz. Diese *Stringkonstante* stellt die zum Abschluss des Countdowns auszugebende Meldung dar. Anschließend wird die Funktion `exit()` verwendet, um `dtrace` zu beenden und zur Shell-Eingabeaufforderung zurückzukehren.

Wenn Sie nun auf die Struktur von `countdown.d` zurückblicken, erkennen Sie, wie durch die Erstellung von zwei Klauseln mit derselben Prüfpunktbeschreibung, aber unterschiedlichen Prädikaten und Aktionen effektiv ein logischer Fluss erzeugt wurde:

i = 10
einmal pro Sekunde,
wenn i größer als Null,
dann trace(i--);
anderenfalls, wenn i gleich Null,
dann trace("blastoff!");
; exit(0);

Wenn Sie komplexe Programme mit Prädikaten schreiben möchten, bietet es sich an, den Algorithmus zunächst auf diese Weise darzustellen. Setzen Sie dann jeden Bestandteil des bedingten Konstrukts in eine separate Klausel mit Prädikat um.

Wir werden jetzt Prädikate mit einem neuen Provider, `syscall`, kombinieren und ein erstes wirkliches Ablaufverfolgungsprogramm in D schreiben. Mit dem Provider `syscall` lassen sich Prüfpunkte zu Beginn oder nach Abschluss eines jeden Solaris-Systemaufrufs aktivieren. Im nächsten Beispiel wird mit DTrace jeder `read(2)` bzw.·`write(2)`-Systemaufruf Ihrer Shell beobachtet. Öffnen Sie zunächst zwei Terminalfenster - eines für DTrace und das andere für den Shell-Prozess, den Sie überwachen möchten. Geben Sie in das zweite Fenster den folgenden Befehl zur Ermittlung der Prozess-ID dieser Shell ein:

```
# echo $$
12345
```

Schreiben Sie nun das nachfolgende D-Programm in das erste Terminalfenster und speichern Sie es unter dem Namen `rw.d`. Ersetzen Sie bei der Eingabe des Programms *12345* durch die Prozess-ID der Shell, die Sie als Antwort auf den Befehl `echo` erhalten haben.

```
syscall::read:entry,
syscall::write:entry
/pid == 12345/
{

}
```

Beachten Sie, dass der Rumpf der Prüfpunktklausel in `rw.d` leer bleibt, da das Programm nur zur Verfolgung von Benachrichtigungen über Prüfpunktauslösungen und nicht zum Aufzeichnen zusätzlicher Informationen vorgesehen ist. Wenn Sie `rw.d` fertig eingegeben haben, starten Sie das Experiment mit `dtrace` und wechseln zum zweiten Shell-Fenster. Geben Sie dort einige Befehle ein und drücken Sie nach jedem Befehl die Eingabetaste. Während Sie die Befehle eingeben, meldet `dtrace` im ersten Fenster die Prüfpunktauslösungen, etwa wie in diesem Beispiel:

```
# dtrace -s rw.d
dtrace: script 'rw.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
    0     34                       write:entry
    0     32                        read:entry
    0     34                       write:entry
    0     32                        read:entry
    0     34                       write:entry
    0     32                        read:entry
    0     34                       write:entry
    0     32                        read:entry
...
```

Sie beobachten gerade, wie die Shell mit `read(2)` - und `write(2)` -Systemaufrufen Zeichen aus dem Terminalfenster liest und das Ergebnis zurückgibt! In diesem Beispiel sind viele der bisher betrachteten Konzepte enthalten - und auch einige neue. Zunächst wird in dem Skript eine einzige Prüfpunktklausel mit mehreren Prüfpunktbeschreibungen verwendet, um `read(2)` und `write(2)` auf gleiche Weise zu instrumentieren. Dabei werden die Beschreibungen wie folgt durch Kommas getrennt:

```
syscall::read:entry,
syscall::write:entry
```

Zur besseren Lesbarkeit erhält jede Prüfpunktbeschreibung eine eigene Zeile. Diese Anordnung ist nicht obligatorisch, sorgt aber für übersichtlichere Skripten. Als Nächstes definiert das Skript ein Prädikat, das nur die von Ihrem Shell-Prozess ausgeführten Systemaufrufe herausfiltert:

```
/pid == 12345/
```

In diesem Prädikat kommt die vordefinierte DTrace-Variable `pid` zum Einsatz, deren Auswertung stets die Prozess-ID des Threads ergibt, der den entsprechenden Prüfpunkt ausgelöst hat. DTrace stellt zahlreiche integrierte Variablendefinitionen für nützliche Informationen wie die Prozess-ID bereit. Sehen Sie hier eine Liste einiger DTrace-Variablen, die Sie in Ihren ersten D-Programmen verwenden können:

Variablenname	Datentyp	Bedeutung
`errno`	`int`	Aktueller `errno`-Wert für Systemaufrufe
`execname`	`string`	Name der ausführbaren Datei des aktuellen Prozesses
`pid`	`pid_t`	Prozess-ID des aktuellen Prozesses
`tid`	`id_t`	Thread-ID des aktuellen Threads
`probeprov`	`string`	Providerfeld der aktuellen Prüfpunktbeschreibung
`probemod`	`string`	Modulfeld der aktuellen Prüfpunktbeschreibung
`probefunc`	`string`	Funktionsfeld der aktuellen Prüfpunktbeschreibung
`probename`	`string`	Namensfeld der aktuellen Prüfpunktbeschreibung

Ändern Sie nun in Ihrem Instrumentationsprogramm versuchsweise die Prozess-ID und die instrumentierten Systemaufruf-Prüfpunkte ab, um das Programm auf andere auf dem System laufende Prozesse anzuwenden. Anschließend können Sie `rw.d` durch eine weitere kleine Änderung in eine sehr einfache Version eines Tracing-Tools für Systemaufrufe wie`truss(1)`. umwandeln. Ein leeres Feld in einer Prüfpunktbeschreibung fungiert als Platzhalter, mit dem alle Prüfpunkte als Übereinstimmung interpretiert werden. Ändern Sie das Programm also auf den folgenden neuen Quellcode ab, sodass *sämtliche* von der Shell ausgeführten Systemaufrufe verfolgt werden:

```
syscall::: entry
/pid == 12345/
{

}
```

Geben Sie einige Befehle wie beispielsweise `cd`, `ls` oder `date` in die Shell ein und beobachten Sie die Ausgabe des DTrace-Programms.

Formatierung der Ausgabe

Die Ablaufverfolgung von Systemaufrufen ist eine sehr wirkungsvolle Methode zur Beobachtung des Verhaltens der meisten Benutzerprozesse. Wenn Sie das Solaris-Dienstprogramm `truss(1)` als Administrator oder Entwickler bereits verwendet haben, wissen Sie wahrscheinlich, dass es im Problemfall immer einmal nützlich sein kann. Wenn Sie `truss` noch nie eingesetzt haben, sollten Sie es jetzt versuchen, indem Sie diesen Befehl in eine Ihrer Shells eingeben:

```
$ truss date
```

Das Ergebnis ist ein formatiertes Ablaufprotokoll aller von `date(1)` ausgeführten Systemaufrufe, gefolgt von einer einzeiligen Befehlsausgabe. Das folgende Beispiel ist eine im Hinblick auf die Formatierung der Ausgabe verbesserte Version des vorherigen Programms `rw.d`. Die an `truss(1)` angelehnte Ausgabe ist nun leichter verständlich. Geben Sie das folgende Programm ein und speichern Sie es unter dem Namen `trussrw.d` :

BEISPIEL 1–2 `trussrw.d`: Ablaufverfolgung von Systemaufrufen mit dem Ausgabeformat von `truss`(1)

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
    printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
}

syscall::read:return,
syscall::write:return
/pid == $1/
{
    printf("\t\t = %d\n", arg1);
}
```

Hier wurde die Konstante `12345` in jedem Prädikat durch den Bezeichner `$1` ersetzt. Über diesen Bezeichner kann der gewünschte Prozess dem Skript als *Argument* übergeben werden: `$1` wird bei der Kompilierung des Skripts durch den Wert des ersten Arguments ersetzt. Zum Ausführen von `trussrw.d` verwenden Sie die `dtrace`-Optionen `-q` und `-s`, gefolgt von der Prozess-ID der Shell als abschließendes Argument. Die Option `-q` gibt an, dass `dtrace` mit minimaler Ausgabe ausgeführt werden und sowohl die Kopfzeile als auch die CPU- und ID-Spalten aus den vorangehenden Beispielen unterdrücken soll. Sie sehen also nur die Ausgabe für die Daten, die ausdrücklich verfolgt werden sollten. Geben Sie den folgenden Befehl ein (wobei Sie `12345` durch die Prozess-ID eines Shell-Prozesses ersetzen) und drücken Sie in der angegebenen Shell wiederholt die Eingabetaste:

```
# dtrace -q -s trussrw.d 12345
                      = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)                = 1
write(2, 0x8089e48,    1)               = 1
read(63, 0x8090a38, 1024)               = 0
read(63, 0x8090a38, 1024)               = 0
write(2, 0x8089e48,   52)               = 52
read(0, 0x8089878,    1)                = 1
write(2, 0x8089e48,    1)               = 1
```

```
read(63, 0x8090a38, 1024)                = 0
read(63, 0x8090a38, 1024)                = 0
write(2, 0x8089e48,   52)                = 52
read(0, 0x8089878,    1)^C
#
```

Nun werden wir uns das D-Programm und seine Ausgabe genauer betrachten. Zuerst wird mit einer Klausel wie im vorigen Programm jeder Aufruf von `read(2)` und `write(2)` der Shell instrumentiert. In diesem Beispiel verwenden wir jedoch die neue Funktion `printf()`, um Daten zu verfolgen und diese in einem bestimmten Format auszugeben:

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
    printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
}
```

Die Funktion `printf()` bietet zwei Fähigkeiten in Einem: Ähnlich wie die bereits verwendete Funktion `trace()` verfolgt sie Daten und zusätzlich kann sie die Daten und anderen Text in einem spezifischen, von Ihnen beschriebenen Format ausgeben. Die Funktion `printf()` weist DTrace an, die jedem Argument nach dem ersten Argument zugehörigen Daten zu verfolgen und die Ergebnisse gemäß den mit dem ersten `printf()`-Argument, der *Formatzeichenkette*, beschriebenen Regeln auszugeben.

Die Formatzeichenkette ist eine normale Zeichenkette (string). Sie kann beliebig viele mit dem Zeichen `%` angeführte Formatumwandlungen enthalten, die beschreiben, wie das entsprechende Argument formatiert werden soll. Die erste Konvertierung in der Formatzeichenkette bezieht sich auf das zweite Argument von `printf()`, die zweite Konvertierung auf das dritte Argument und so weiter. Text zwischen den Umwandlungen wird wörtlich wiedergegeben. Das auf das `%`-Umwandlungszeichen folgende Zeichen beschreibt das für das entsprechende Argument zu verwendende Format. Die drei Formatumwandlungen in `trussrw.d` haben folgende Bedeutung:

`%d`	Der entsprechende Wert wird als Dezimalzahl ausgegeben.
`%s`	Der entsprechende Wert wird als Zeichenkette ausgegeben.
`%x`	Der entsprechende Wert wird als Hexadezimalzahl ausgegeben.

DTrace `printf()` wirkt wie die C-Bibliotheksroutine `printf(3C)` bzw. das Shell-Dienstprogramm `printf(1)`. Falls Ihnen·`printf()` neu ist, klärt Sie Kapitel 12, „Formatierung der Ausgabe“ im Detail über die Formate und Optionen auf. Lesen Sie dieses Kapitel aber auch dann aufmerksam durch, wenn Sie `printf()` bereits aus einer anderen Sprache kennen. `printf()` ist in D integriert und bietet Ihnen einige neue, speziell für DTrace entwickelte Formatumwandlungen.

Zur Unterstützung beim Schreiben fehlerfreier Programme prüft der D-Compiler jede `printf()`-Formatzeichenkette auf ihre Argumentliste. Ändern Sie `probefunc` in der obigen Klausel in die Ganzzahl `123` ab. Wenn Sie das abgeänderte Programm ausführen, erhalten Sie eine Fehlermeldung, die besagt, dass die String-Formatumwandlung `%s` nicht für ein Integer-Argument geeignet ist:

```
# dtrace -q -s trussrw.d
dtrace: failed to compile script trussrw.d: line 4: printf( )
       argument #2 is incompatible with conversion #1 prototype:
            conversion: %s
             prototype: char [] or string (or use stringof)
              argument: int
#
```

Für die Ausgabe der read- oder write-Systemaufrufe und ihrer Argumente verwenden Sie die Anweisung `printf()`:

```
printf("%s(%d, 0x%x, %4d)", probefunc, arg0, arg1, arg2);
```

Damit werden der Name der aktuellen Prüfpunktfunktion und die ersten drei Integer-Argumente des Systemaufrufs, die in den DTrace-Variablen `arg0`, `arg1` und `arg2` zur Verfügung stehen, verfolgt. Weitere Informationen zu Prüfpunktargumenten finden Sie in Kapitel 3, „Variablen“. Das erste Argument von `read(2)` und `write(2)` ist ein als Dezimalzahl dargestellter Dateideskriptor. Das zweite Argument ist eine als Hexadezimalwert formatierte Pufferadresse. Bei dem letzen Argument handelt es sich schließlich um die in Form eines Dezimalwerts wiedergegebene Puffergröße. Die Formatangabe `%4d` für das dritte Argument bedeutet, dass der Wert mit der Formatumwandlung `%d` und einer minimalen Feldbreite von 4 Zeichen dargestellt werden soll. Ist die Ganzzahl weniger als 4 Zeichen lang, fügt `printf()` zur Ausrichtung der Ausgabe zusätzliche Leerzeichen ein.

Um das Ergebnis des Systemaufrufs wiederzugeben und jede Ausgabezeile abzuschließen, verwenden Sie die folgende Klausel:

```
syscall::read:return,
syscall::write:return
/pid == $1/
{
    printf("\t\t = %d\n", arg1);
}
```

Beachten Sie, dass der Provider `syscall` außer `entry` für jeden Systemaufruf auch einen Prüfpunkt namens `return` veröffentlicht. Durch die DTrace-Variable `arg1` für die syscall-Prüfpunkte `return` wird der Rückgabewert des Systemaufrufs eingesetzt. Der Rückgabewert wird als Dezimalzahl formatiert. Die in der Formatzeichenkette mit umgekehrten Schrägstrichen beginnenden Zeichenfolgen erstrecken sich bis zum Tabulator (`\t`) bzw. zur neuen Zeile (`\n`). Diese *„Escape-Folgen“* erleichtern die Wiedergabe oder

Aufzeichnung schwer darzustellender Zeichen. D unterstützt denselben Ersatzdarstellungssatz wie C, C++ und Java. Eine vollständige Liste der Escape-Sequenzen finden Sie in Kapitel 2, „Typen, Operatoren und Ausdrücke“.

Vektoren

D bietet Ihnen die Möglichkeit, Variablen des Typs Integer und anderer Typen zur Darstellung von Zeichenketten sowie gemischte Typen zu definieren, die als *structs* (Strukturen) und *unions* (Unionen) bezeichnet werden. Wenn Sie sich mit der C-Programmierung auskennen, wird es Sie freuen, dass Sie in D dieselben Typen wie in C verwenden können. Wenn Sie kein C-Experte sind, gibt es trotzdem keinen Grund zur Besorgnis: Sämtliche verschiedenen Datentypen werden in Kapitel 2, „Typen, Operatoren und Ausdrücke“ erklärt. Außerdem unterstützt D eine besondere Variablenart, den so genannten *assoziativen Vektor*. Ein assoziativer Vektor gleicht einem einfachen Vektor (Array) insofern, als er Schlüsseln Werte zuordnet. In einem assoziativen Vektor sind die Schlüssel jedoch nicht auf Ganzzahlen eines festgelegten Wertebereichs beschränkt.

Assoziative Vektoren in D lassen sich durch eine Liste von einem oder mehreren Werten beliebigen Typs indizieren. Die Schlüsselwerte bilden zusammen ein *Tupel*, das zur Indizierung im Vektor und zum Zugriff auf den oder Ändern des Werts des entsprechenden Schlüssels eingesetzt wird. Jedes für einen bestimmten assoziativen Vektor verwendetes Tupel muss derselben Typensignatur entsprechen. Das heißt, dass jedes Tupel dieselbe Länge und dieselben Schlüsseltypen in derselben Reihenfolge aufweisen muss. Außerdem sind die den Elementen eines gegebenen assoziativen Vektors zugeordneten Werte für den gesamten Vektor auf einen einzigen Typ festgelegt. Die folgende D-Anweisung definiert beispielsweise einen neuen assoziativen Vektor `a` des Wertetyps `int` mit der Tupelsignatur `[ string, int ]` und speichert den ganzzahligen Wert 456 im Vektor:

```
a["hello", 123] = 456;
```

Auf die Elemente eines definierten Vektors kann wie auf jede andere D-Variable zugegriffen werden. So ändern wir beispielsweise mit der folgenden D-Anweisung das zuvor in `a` gespeicherte Vektorelement, indem wir den Wert von 456 auf 457 erhöhen:

```
a["hello", 123]++;
```

Die Werte aller noch nicht zugewiesenen Vektorelemente werden auf Null gesetzt. Setzen wir nun einen assoziativen Vektor in ein D-Programm ein. Geben Sie das folgende Programm ein und speichern Sie es unter dem Namen `rwtime.d`:

BEISPIEL 1–3 `rwtime.d`: Zeit für `read(2)`- und `write(2)`-Aufrufe

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
```

BEISPIEL 1–3 `rwtime.d`: Zeit für `read(2)`- und `write(2)`-Aufrufe *(Fortsetzung)*

```
{
    ts[probefunc] = timestamp;
}

syscall::read:return,
syscall::write:return
/pid == $1 && ts[probefunc] != 0/
{
    printf("%d nsecs", timestamp - ts[probefunc]);
}
```

Geben Sie bei der Ausführung von `rwtime.d` wie schon bei `trussrw.d` die ID des Shell-Prozesses an. Wenn Sie einige Shell-Befehle eingeben, sehen Sie die Dauer jedes Systemaufrufs. Geben Sie folgenden Befehl ein und drücken Sie anschließend in der anderen Shell mehrmals die Eingabetaste:

```
# dtrace -s rwtime.d ‘pgrep -n ksh‘
dtrace: script ’rwtime.d’ matched 4 probes
CPU     ID                    FUNCTION:NAME
  0     33                      read:return 22644 nsecs
  0     33                      read:return 3382 nsecs
  0     35                     write:return 25952 nsecs
  0     33                      read:return 916875239 nsecs
  0     35                     write:return 27320 nsecs
  0     33                      read:return 9022 nsecs
  0     33                      read:return 3776 nsecs
  0     35                     write:return 17164 nsecs
...
^C
#
```

Um ein Protokoll der für jeden Systemaufruf abgelaufenen Zeit zu erhalten, müssen Sie sowohl den Eintritt in als auch die Rückkehr aus `read(2)` und `write(2)` instrumentieren und die Zeit an jedem Punkt prüfen. Anschließend ist bei der Rückkehr aus einem gegebenen Systemaufruf die Differenz zwischen der ersten und der zweiten Zeitmarke zu berechnen. Hierbei könnten Sie für jeden Systemaufruf eine eigene Variable verwenden, doch das würde die Erweiterung des Programms um zusätzliche Systemaufrufe sehr beschwerlich machen. Einfacher ist es, stattdessen auf einen durch den Namen der Prüfpunktfunktion indizierten assoziativen Vektor zurückzugreifen. Sehen Sie hier die erste Prüfpunktklausel:

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
```

```
    ts[probefunc] = timestamp;
}
```

Diese Klausel definiert einen Vektor namens `ts` und weist dem entsprechenden Element den Wert der DTrace-Variable `timestamp`zu. Diese Variable gibt den Wert eines stets anwachsenden Nanosekundenzählers zurück, ähnlich der Solaris-Bibliotheksroutine`gethrtime(3C)`. Wenn die Zeitmarke des Eintritts gespeichert ist, wird `timestamp` erneut durch den Prüfpunkt für die Rückkehr geprüft, der dann die Differenz zwischen der aktuellen Uhrzeit und dem gespeicherten Wert meldet:

```
syscall::read:return,
syscall::write:return
/pid == $1 && ts[probefunc] != 0/
{
    printf("%d nsecs", timestamp - ts[probefunc]);
}
```

Das Prädikat des `return`-Prüfpunkts setzt voraus, dass DTrace den betreffenden Prozess verfolgt und der entsprechende `entry`-Prüfpunkt bereits ausgelöst wurde und `ts[probefunc]` einen Wert ungleich Null zugewiesen hat. Durch diesen Trick lässt sich eine ungültige Ausgabe beim Start von DTrace vermeiden. Wartet die Shell, wenn Sie `dtrace` ausführen, bereits in einem `read(2)`-Systemaufruf auf eine Eingabe, wird der Prüfpunkt `read:return` ohne einen vorangehenden `read:entry` für diesen ersten `read(2)`-Aufruf ausgelöst, und die Prüfung von `ts[probefunc]` ergibt Null, da noch keine Zuweisung stattgefunden hat.

Externe Symbole und Typen

Die DTrace-Instrumentation wird innerhalb des Solaris-Betriebssystemkerns ausgeführt. Das heißt, dass neben den speziellen DTrace-Variablen und Prüfpunktargumenten auch auf Kerneldatenstrukturen, Symbole und Typen zugegriffen werden kann. Dank dieser Fähigkeiten können fortgeschrittene DTrace-Benutzer, Administratoren, Wartungstechniker und Treiberentwickler das Verhalten des Betriebssystemkernels und der Gerätetreiber auf den unteren Ebenen beobachten. In den Literaturangaben zu Beginn dieses Dokuments finden Sie Titel zu den Interna des Betriebssystems Solaris.

In D kommt das Backquote-Zeichen (oder Accent grave) (`) als spezieller Bereichsoperator zum Ansprechen von Symbolen zum Einsatz, die zwar im Betriebssystem, nicht aber im D-Programm definiert sind. So enthält der Solaris-Kernel beispielsweise eine C-Deklaration eines über das System abstimmbaren Parameters (Tunables) namens `kmem_flags` zum Aktivieren von Debugging-Leistungsmerkmalen für die Speicherzuweisung. Im *Solaris Tunable Parameters Reference Manual* finden Sie weitere Informationen zu `kmem_flags`. Dieses Tunable ist im Kernel-Quellcode wie folgt in C deklariert:

```
int kmem_flags;
```

Mit folgender D-Anweisung lässt sich der Wert dieser Variable in einem D-Programm verfolgen:

```
trace(‘kmem_flags);
```

DTrace weist jedem Kernelsymbol den entsprechenden Typ aus dem C-Betriebssystemcode zu und bietet dadurch einen einfachen quellcodebasierten Zugriff auf die nativen Datenstrukturen des Betriebssystems. Die Namen von Kernelsymbolen werden in einem von den D-Variablen- und Funktionsbezeichnern getrennten Namensraum gehalten, sodass keine Gefahr eines Konflikts zwischen diesen Namen und Ihren D-Variablen besteht.

Sie haben nun Ihre Blitztour durch DTrace beendet und viele der grundlegenden DTrace-Bausteine kennen gelernt, die zum Schreiben größerer, komplexerer D-Programme benötigt werden. In den nachfolgenden Kapiteln wird der vollständige Regelsatz für D beschrieben und erläutert, wie DTrace komplexe Performancemessungen und Funktionsanalysen des Systems zu einer leichten Angelegenheit macht. Später erfahren Sie, wie Sie mit DTrace das Verhalten von Benutzeranwendungen mit dem Systemverhalten verbinden können und sich so der gesamte Software-Stack analysieren lässt.

Sie haben gerade erst begonnen!

◆ ◆ ◆ KAPITEL 2

2 Typen, Operatoren und Ausdrücke

Mit D können Sie auf zahlreiche Datenobjekte zugreifen und sie manipulieren: Variablen und Datenstrukturen lassen sich erstellen und ändern, im Betriebssystemkernel und in Benutzerprozessen definierte Datenobjekte können angesteuert und Integer-, Fließkomma- und Stringkonstanten deklariert werden. D bietet eine Obergruppe der ANSI-C-Operatoren, die für die Manipulation von Objekten und zum Erstellen komplexer Ausdrücke dienen. In diesem Kapitel wird der Regelsatz für Typen, Operatoren und Ausdrücke in allen Einzelheiten beschrieben.

Bezeichnernamen und Schlüsselwörter

In D bestehen Bezeichnernamen aus Groß- und Kleinbuchstaben, Ziffern und Unterstrichen. Das erste Zeichen muss ein Buchstabe oder Unterstrich sein. Alle mit einem Unterstrich (_) beginnenden Bezeichnernamen sind den D-Systembibliotheken vorbehalten. Verzichten Sie in Ihren D-Programmen auf solche Namen. Vereinbarungsgemäß verwenden D-Programmierer normalerweise Namen mit Groß- und Kleinbuchstaben für Variablen und Namen in ausschließlich Großbuchstaben für Konstanten.

Die D-Schlüsselwörter stellen spezielle, zur Verwendung in der Programmiersprachensyntax selbst reservierte Bezeichner dar. Diese Namen werden stets in Kleinbuchstaben angegeben und dürfen nicht als Namen für D-Variablen verwendet werden.

TABELLE 2–1 D-Schlüsselwörter

`auto`*	`goto`*	`sizeof`
`break`*	`if`*	`static`*
`case`*	`import`*+	`string`+
`char`	`inline`	`stringof`+

TABELLE 2–1 D-Schlüsselwörter *(Fortsetzung)*

`const`	`int`	`struct`
`continue`*	`long`	`switch`*
`counter`*+	`offsetof`+	`this`+
`default`*	`probe`*+	`translator`+
`do`*	`provider`*+	`typedef`
`double`	`register`*	`union`
`else`*	`restrict`*	`unsigned`
`enum`	`return`*	`void`
`extern`	`self`+	`volatile`
`float`	`short`	`while`*
`for`*	`signed`	`xlate`+

In D ist eine Obergruppe der ANSI-C-Schlüsselwörter für die Verwendung als Schlüsselwörter reserviert. Die zur künftigen Nutzung durch D reservierten Schlüsselwörter sind mit „*" gekennzeichnet. Wenn Sie versuchen, ein zur künftigen Nutzung vorgesehenes Schlüsselwort einzusetzen, generiert der D-Compiler einen Syntaxfehler. Die in D, aber nicht in ANSI-C definierten Schlüsselwörter, sind mit „+" gekennzeichnet. D umfasst den vollständigen Satz der ANSI-C-Typen und -Operatoren. Der wesentliche Unterschied zur Programmierung in D besteht im Verzicht auf Kontrollstrukturen. Die in ANSI-C als Kontrollstrukturen dienenden Schlüsselwörter sind in D zur künftigen Verwendung reserviert.

Datentypen und -größen

In D stehen Ihnen die Grunddatentypen für Ganzzahlen- und Fließpunktkonstanten zur Verfügung. In D-Programmen können arithmetische Operationen nur an Ganzzahlen vorgenommen werden. Gleitkommakonstanten sind zum Initialisieren von Datenstrukturen zulässig, aber die Gleitkommaarithmetik ist in D nicht erlaubt. D stellt zum Schreiben von Programmen ein 32-Bit- und 64-Bit-Datenmodell zur Verfügung. Bei der Ausführung des Programms wird das native Datenmodell des aktiven Betriebssystemkernels verwendet. Mit `isainfo -b` lässt sich das native Datenmodell des jeweiligen Systems ermitteln.

Die folgende Tabelle enthält die Namen der Integer-Typen und ihre Größe in beiden Datenmodellen. In der nativen Byte-Kodierung des Systems werden Integer-Typen stets als Zweierkomplement dargestellt.

TABELLE 2–2 Integer-Datentypen in D

Typname	32-Bit-Größe	64-Bit-Größe
`char`	1 Byte	1 Byte
`short`	2 Byte	2 Byte
`int`	4 Byte	4 Byte
`long`	4 Byte	8 Byte
`long long`	8 Byte	8 Byte

Integer-Typen können das Präfix `signed` (mit Vorzeichen) oder `unsigned` (ohne Vorzeichen) erhalten. Wenn kein Vorzeichen-Kennzeichner vorhanden ist, gilt der Datentyp als vorzeichenbehaftet. Der D-Compiler stellt außerdem die Typ-Aliasnamen in folgender Tabelle bereit:

TABELLE 2–3 Aliasnamen für Integer-Typen in D

Typname	Beschreibung
`int8_t`	1-Byte-Integer mit Vorzeichen
`int16_t`	2-Byte-Integer mit Vorzeichen
`int32_t`	4-Byte-Integer mit Vorzeichen
`int64_t`	8-Byte-Integer mit Vorzeichen
`intptr_t`	Integer mit Vorzeichen der Größe eines Zeigers
`uint8_t`	1-Byte-Integer ohne Vorzeichen
`uint16_t`	2-Byte-Integer ohne Vorzeichen
`uint32_t`	4-Byte-Integer ohne Vorzeichen
`uint64_t`	8-Byte-Integer ohne Vorzeichen
`uintptr_t`	Integer ohne Vorzeichen der Größe eines Zeigers

Diese Typ-Aliasnamen haben die gleiche Wirkung wie die Namen des entsprechenden Grundtyps in der vorigen Tabelle und sind für jedes Datenmodell angemessen definiert. So ist beispielsweise der Typname `uint8_t` ein Aliasname für den Typ `unsigned char`. Wie Sie eigene Typ-Aliasnamen für Ihre D-Programme definieren können, erfahren Sie in Kapitel 8, „Typ- und Konstantendefinitionen“.

Aus Gründen der Kompatibilität mit ANSI-C-Deklarationen und -Typen bietet D auch Gleitkommatypen. Gleitkomma-Operatoren werden in D nicht unterstützt. Es ist allerdings

möglich, Gleitkomma-Datenobjekte mit der Funktion `printf()` zu verfolgen und zu formatieren. Die Gleitkommatypen in der folgenden Tabelle sind zulässig:

TABELLE 2–4 Gleitkomma-Datentypen in D

Typname	32-Bit-Größe	64-Bit-Größe
`float`	4 Byte	4 Byte
`double`	8 Byte	8 Byte
`long double`	16 Byte	16 Byte

Außerdem steht in D der spezielle Typ `string` zur Darstellung von ASCII-Zeichenketten zur Verfügung. Zeichenketten werden in Kapitel 6, „Zeichenketten“ näher erläutert.

Konstanten

Integer-Konstanten (Ganzzahlkonstanten) können in dezimaler (`12345`), oktaler (`012345`) oder hexadezimaler (`0x12345`) Form geschrieben werden. Oktale Konstanten (Grundzahl 8) müssen eine Null als Präfix erhalten. Hexadezimalen Konstanten (Grundzahl 16) ist entweder `0x` oder `0X` voranzustellen. Integer-Konstanten wird der kleinste der Datentypen `int`, `long` oder `long long` zugewiesen, mit dem ihr Wert dargestellt werden kann. Bei negativen Werten wird die vorzeichenbehaftete Version des entsprechenden Typs verwendet. Positive Werte, die zu groß für den Typ mit Vorzeichen sind, werden durch den entsprechenden vorzeichenlosen Typ dargestellt. An jede Integer-Konstante kann zur expliziten Angabe des D-Typs einer der folgenden Zusätze angehängt werden:

`u` oder `U`	`unsigned`-Version des vom Compiler ausgewählten Typs
`l` oder `L`	`long`
`ul` oder `UL`	`unsigned long`
`ll` oder `LL`	`long long`
`ull` oder `ULL`	`unsigned long long`

Gleitkommakonstanten werden immer in dezimaler Form dargestellt und müssen entweder ein Dezimaltrennzeichen (`12.345`), einen Exponent (`123e45`) oder beides enthalten (`123.34e-5`). Standardmäßig wird Gleitkommakonstanten der Typ `double` zugewiesen. An jede Gleitkommakonstante kann zur expliziten Angabe des D-Typs einer der folgenden Zusätze angehängt werden:

f oder F	float
l oder L	long double

Zeichenkonstanten werden als einzelnes Zeichen oder in ihrer Ersatzdarstellung zwischen einfachen Anführungszeichen dargestellt (`'a'`). Zeichenkonstanten wird der Typ `int` zugewiesen und sie sind gleichbedeutend mit Integer-Konstanten, deren Wert durch den Wert des entsprechenden Zeichens im ASCII-Zeichensatz bestimmt ist. Eine Liste von Zeichen und ihren Werten finden Sie unter `ascii(5)`. Auch alle speziellen Ersatzdarstellungen in der nachfolgenden Tabelle sind in Zeichenkonstanten erlaubt. D unterstützt dieselben Ersatzdarstellungen, wie sie in ANSI-C verwendet werden.

TABELLE 2–5 Ersatzdarstellungen für Zeichen in D

`\a`	Klingelzeichen	`\\`	Gegenschrägstrich
`\b`	Rückschritt (Backspace)	`\?`	Fragezeichen
`\f`	Seitenvorschub	`\'`	(einfaches) Anführungszeichen
`\n`	Zeilentrenner	`\"`	(doppeltes) Anführungszeichen
`\r`	Wagenrücklauf	`\0`*oo*	Oktalwert 0*oo*
`\t`	(waagerechter) Tabulator	`\x`*hh*	Hexadezimalwert 0x*hh*
`\v`	Vertikal-Tabulator	`\0`	Nullzeichen

Indem Sie mehrere Zeichenangaben zwischen einfache Anführungszeichen setzen, können Sie Ganzzahlen erstellen, deren einzelne Byte je nach der entsprechenden Zeichenangabe initialisiert werden. Die Byte werden ab der Zeichenkonstante von links nach rechts gelesen und der entstehenden Ganzzahl entsprechend der nativen Byte-Reihenfolge des Betriebssystems in der Darstellung Big Endian oder Little Endian zugeordnet. Eine Zeichenkonstante fasst bis zu acht Zeichenangaben.

Durch Schreibung zwischen doppelten Anführungszeichen lassen sich Zeichenkettenkonstanten einer beliebigen Länge zusammensetzen (`"hello"`). Zeichenkettenkonstanten dürfen keine wörtlichen Zeilentrenner enthalten. Zur Angabe einer neuen Zeile innerhalb einer Zeichenkette verwenden Sie stattdessen die Ersatzdarstellung `\n`. Alle bereits für Zeichenkonstanten aufgeführten Sonderzeichen-Ersatzdarstellungen dürfen auch in Zeichenkettenkonstanten vorkommen. Ähnlich wie in ANSI-C werden Zeichenketten als Vektoren von Zeichen mit einem abschließenden Nullzeichen (`\0`) dargestellt, das implizit an jede deklarierte Zeichenkettenkonstante angefügt wird. Zeichenkettenkonstanten wird der spezielle D-Typ `string` zugewiesen. Der D-Compiler verfügt über besondere Leistungsmerkmale zum Vergleichen und Verfolgen von Zeichenvektoren, die als Zeichenketten deklariert wurden. Mehr darüber erfahren Sie inKapitel 6, „Zeichenketten“.

Arithmetische Operatoren

D stellt Ihnen für Ihre Programme die binären arithmetischen Operatoren in der nachfolgenden Tabelle zur Verfügung. Diese Operatoren haben dieselbe Bedeutung für Ganzzahlen wie in ANSI-C.

TABELLE 2–6 Binäre arithmetische Operatoren in D

`+`	Addition von Ganzzahlen
`-`	Subtraktion von Ganzzahlen
`*`	Multiplikation von Ganzzahlen
`/`	Division von Ganzzahlen
`%`	Restwert von Ganzzahlen

Arithmetische Operationen sind in D ausschließlich an Ganzzahlen-Operanden oder, wie in Kapitel 5, „Zeiger und Vektoren“ erläutert, an Zeigern möglich. Auf Gleitkomma-Operanden lassen sich arithmetische Operationen in D-Programmen nicht anwenden. Die DTrace-Ausführungsumgebung nimmt keinerlei Einfluss auf Integer-Überläufe oder -Unterläufe. Bei bestehendem Über- und Unterlaufrisiko müssen Sie selbst auf diese Bedingungen prüfen.

Die DTrace-Ausführungsumgebung prüft automatisch auf Divisionen durch Null und meldet derartige Fehler, die durch eine falsche Verwendung der Operatoren `/` und `%` entstehen. Wenn ein D-Programm eine ungültige Division durchführt, deaktiviert DTrace die betreffende Instrumentation automatisch und meldet den Fehler. Von DTrace erkannte Fehler haben keinerlei Auswirkungen auf andere DTrace-Benutzer oder den Betriebssystemkernel. Sollte Ihr D-Programm einen dieser Fehler enthalten, müssen Sie sich also keine Sorgen machen, dass es Schaden anrichten könnte.

Neben diesen binären Operatoren sind `+` und `-` auch als unäre Operatoren verwendbar. Sie haben eine höhere Priorität als alle binären arithmetischen Operatoren. Die Eigenschaften in Bezug auf Rangfolge und Assoziativität aller D-Operatoren wird in Tabelle 2–11 dargestellt. Die Rangfolge lässt sich durch Zusammenfassen von Ausdrücken in Klammern `( )` beeinflussen.

Relationale Operatoren

D stellt Ihnen für Ihre Programme die binären relationalen Operatoren (Vergleichsoperatoren) in folgender Tabelle zur Verfügung. Diese Operatoren haben dieselbe Bedeutung wie in ANSI-C.

TABELLE 2–7 Relationale Operatoren in D

`<`	Operand auf linker Seite ist kleiner als Operand auf rechter Seite
`<=`	Operand auf linker Seite ist kleiner als oder gleich dem Operanden auf rechter Seite
`>`	Operand auf linker Seite ist größer als Operand auf rechter Seite
`>=`	Operand auf linker Seite ist größer als oder gleich dem Operanden auf rechter Seite
`==`	Operand auf linker Seite ist gleich dem Operanden auf rechter Seite
`!=`	Operand auf linker Seite ist ungleich dem Operanden auf rechter Seite

Relationale Operatoren werden meistens zum Schreiben von D-Prädikaten eingesetzt. Jeder Operator ergibt einen Wert des Typs `int`, der gleich 1 ist, wenn die Bedingung wahr ist und Null, wenn sie falsch ist.

Relationale Operatoren können auf Paare von Ganzzahlen, Zeigern oder Zeichenketten angewendet werden. Beim Vergleich von Zeigern ist das Ergebnis gleichbedeutend mit einem Integer-Vergleich der zwei als vorzeichenlose Ganzzahlen interpretierten Zeiger. Das Ergebnis eines Vergleichs von Zeichenketten wird wie durch Anwendung von `strcmp(3C)` auf die beiden Operanden ermittelt. Sehen Sie hier einige Beispiele für Zeichenkettenvergleiche in D und ihre Ergebnisse:

`"coffee" < "espresso"`	... gibt 1 (wahr) zurück
`"coffee" == "coffee"`	... gibt 1 (wahr) zurück
`"coffee" >= "mocha"`	... gibt 0 (falsch) zurück

Relationale Operatoren können auch zum Vergleichen eines einem Aufzählungstyp zugewiesenen Datenobjekts mit einem beliebigen der Enumerator-Tags verwendet werden, die im Aufzählungstyp definiert sind. Aufzählungstypen ermöglichen die Erstellung benannter Ganzzahlkonstanten. Kapitel 8, „Typ- und Konstantendefinitionen“ befasst sich ausführlich mit Aufzählungen.

Logische Operatoren

D stellt Ihnen für Ihre Programme die folgenden binären logischen Operatoren zur Verfügung. Die ersten beiden Operatoren sind gleichbedeutend mit den entsprechenden ANSI-C-Operatoren.

TABELLE 2–8 Logische Operatoren in D

&&	Logisches *UND*: wahr, wenn beide Operanden wahr sind
||	Logisches *ODER*: wahr, wenn einer oder beide Operanden wahr sind
^^	Logisches *EXKLUSIV-ODER*: wahr, wenn genau ein Operand wahr ist

Logische Operatoren werden meistens zum Schreiben von D-Prädikaten eingesetzt. Der logische UND-Operator führt eine Short-Circuit-Evaluation durch: Wenn der Operand auf der linken Seite falsch ist, wird der weiter rechts stehende Ausdruck nicht mehr ausgewertet. Auch der logische ODER-Operator führt eine Short-Circuit-Evaluation durch: Wenn der Operand auf der linken Seite wahr ist, wird der weiter rechts stehende Ausdruck nicht mehr ausgewertet. Beim logischen EXKLUSIV-ODER-Operator erfolgt die Auswertung ohne Short-Circuit: Es werden stets beide Operanden des Ausdrucks ausgewertet.

Zusätzlich zu den binären logischen Operatoren steht der unäre `!`-Operator für logische Negierungen eines einzelnen Operanden zur Verfügung: Er konvertiert einen Operanden gleich Null in Eins und einen Operanden ungleich Null in eine Null. D-Programmierer verwenden `!` konventionsgemäß beim Umgang mit Ganzzahlen, die boolesche Werte darstellen sollen, und `== 0` bei nicht-booleschen Ganzzahlen. Die beiden Ausdrücke sind jedoch eigentlich bedeutungsgleich.

Die logischen Operatoren können auf Operanden der Integer- oder Zeigertypen angewendet werden. Zeiger-Operanden werden von den logischen Operatoren als ganzzahlige Werte ohne Vorzeichen interpretiert. Wie bei allen logischen und relationalen Operatoren in D sind auch hier die Operanden wahr, wenn sie einen ganzzahligen Wert ungleich Null und falsch, wenn sie einen ganzzahligen Wert gleich Null aufweisen.

Bitweise Operatoren

D bietet die folgenden binären Operatoren zur Manipulation einzelner Bits innerhalb von numerischen Operanden. Diese Operatoren haben dieselbe Bedeutung wie in ANSI-C.

TABELLE 2–9 Bitweise Operatoren in D

&	Bitweises UND
|	Bitweises ODER
^	Bitweises EXKLUSIV-ODER
<<	Operand auf linker Seite um die mit dem Operanden auf der rechten Seite angegebene Bitanzahl nach links verschieben

TABELLE 2–9 Bitweise Operatoren in D *(Fortsetzung)*

>>	Operand auf linker Seite um die mit dem Operanden auf der rechten Seite angegebene Bitanzahl nach rechts verschieben

Der binäre Operator & dient zum Löschen von Bits aus einem ganzzahligen Operanden. Der binäre Operator | dient zum Einsetzen von Bits in einen ganzzahligen Operanden. Der binäre Operator ^ gibt an jeder Bitposition eine 1 zurück, an der exakt eines der entsprechenden Operand-Bits gesetzt ist.

Mit den Schiebeoperatoren lassen sich Bits innerhalb eines gegebenen Integer-Operanden nach links oder rechts verschieben. Durch eine Linksverschiebung werden leere Bitpositionen auf der rechten Seite des Ergebnisses mit Nullen gefüllt. Durch eine Rechtsverschiebung mit vorzeichenlosem Integer-Operanden werden leere Bitpositionen auf der linken Seite des Ergebnisses mit Nullen gefüllt. Bei einer Rechtsverschiebung mit vorzeichenbehaftetem Integer-Operanden werden leere Bitpositionen auf der linken Seite mit dem Wert des Vorzeichenbits gefüllt. Dies wird auch als *arithmetische Verschiebung* bezeichnet.

Die Verschiebung eines ganzzahligen Werts um eine negative Bitanzahl oder eine Bitanzahl, die größer ist als die Anzahl der Bits im linken Operanden selbst, ergibt ein unbestimmtes Resultat. Wenn diese Bedingung beim Kompilieren des D-Programms vom D-Compiler erkannt wird, generiert dieser eine Fehlermeldung.

Zusätzlich zu den binären logischen Operatoren steht der unäre ~-Operator für bitweise Negierungen eines einzelnen Operanden zur Verfügung: Er konvertiert jedes Nullbit im Operanden in ein 1-Bit und jedes 1-Bit im Operanden in ein Nullbit.

Zuweisungsoperatoren

D bietet die folgenden binären Zuweisungsoperatoren zum Ändern von D-Variablen. Nur D-Variablen und Vektoren lassen sich ändern. Kerneldatenobjekte und Konstanten können mit den D-Zuweisungsoperatoren nicht geändert werden. Die Zuweisungsoperatoren haben dieselbe Bedeutung wie in ANSI-C.

TABELLE 2–10 Zuweisungsoperatoren in D

=	Linken Operanden mit rechtem Ausdruckswert gleichsetzen
+=	Linken Operanden um den rechten Ausdruckswert erhöhen
-=	Linken Operanden um den rechten Ausdruckswert verringern
*=	Linken Operanden mit dem rechten Ausdruckswert multiplizieren
/=	Linken Operanden durch den rechten Ausdruckswert dividieren

TABELLE 2–10 Zuweisungsoperatoren in D *(Fortsetzung)*

`%=`	Rest des linken Operanden mit dem rechten Ausdruckswert als Divisor errechnen
`|=`	Bitweise ODER-Operation an linkem Operanden mit rechtem Ausdruckswert durchführen
`&=`	Bitweise UND-Operation an linkem Operanden mit rechtem Ausdruckswert durchführen
`^=`	Bitweise EXKLUSIV-ODER-Operation an linkem Operanden mit rechtem Ausdruckswert durchführen
`<<=`	Operanden auf linker Seite um die mit dem Ausdruckswert auf der rechten Seite angegebene Bitanzahl nach links verschieben
`>>=`	Operanden auf linker Seite um die mit dem Ausdruckswert auf der rechten Seite angegebene Bitanzahl nach rechts verschieben

Alle Zuweisungsoperatoren außer = dienen als Kurzform für die Verwendung des Operators = mit einem der weiter oben beschriebenen Operatoren. So ist beispielsweise der Ausdruck `x = x + 1` gleichwertig mit dem Ausdruck `x += 1`, bis auf die Tatsache, dass der Ausdruck `x` einmal ausgewertet wird. Diese Zuweisungsoperatoren unterliegen denselben Regeln für Operandentypen wie die bereits beschriebenen binären Formen.

Das Ergebnis jedes Zuweisungsoperators ist ein Ausdruck gleich dem neuen Wert des Ausdrucks auf der linken Seite. Sie können die Zuweisungsoperatoren sowie alle bisher genannten Operatoren kombinieren, um Ausdrücke von beliebiger Komplexität zu formen. Terme in komplexen Ausdrücken können durch Klammern `( )` gruppiert werden.

Inkrement- und Dekrement-Operatoren

In D stehen Ihnen die speziellen unären Operatoren `++` und `--` zum Erhöhen und Verringern von Zeigern und Ganzzahlen zur Verfügung. Diese Operatoren haben dieselbe Bedeutung wie in ANSI-C. Sie sind nur auf Variablen anwendbar und können vor oder nach dem Variablennamen stehen. Wenn der Operator vor dem Variablennamen steht, wird zuerst die Variable geändert und der resultierende Ausdruck dann dem neuen Wert der Variable gleichgesetzt. Beispielsweise ergeben die zwei folgenden Ausdrücke identische Resultate:

```
x += 1;
y = x;
```

```
y = ++x;
```

Steht der Operator nach dem Variablennamen, dann wird die Variable erst verändert, nachdem ihr aktueller Wert zur Verwendung im Ausdruck zurückgegeben wurde. Beispielsweise ergeben die zwei folgenden Ausdrücke identische Resultate:

```
y = x;                                  y = x--;
x -= 1;
```

Mit den Inkrement- und Dekrement-Operatoren können Sie neue Variablen erstellen, ohne sie zu deklarieren. Wenn Sie eine Variablendeklaration unterlassen und den Inkrement- oder Dekrement-Operator auf eine Variable anwenden, wird die Variable implizit als Typ `int64_t` erklärt.

Die Inkrement- und Dekrement-Operatoren lassen sich auf Ganzzahl- oder Zeigervariablen anwenden. Bei Anwendung auf Ganzzahlvariablen erhöhen bzw. verringern die Operatoren den entsprechenden Wert um 1. Werden sie auf Zeigervariablen angewendet, erhöhen bzw. verringern die Operatoren die Zeigeradresse um die Größe des mit dem Zeiger referenzierten Datentyps. Zeiger und Zeigerarithmetik in D werden in Kapitel 5, „Zeiger und Vektoren“ behandelt.

Bedingte Ausdrücke

D unterstützt zwar keine if-then-else-Konstrukte, bietet aber Unterstützung für einfache bedingte Ausdrücke unter Verwendung der Operatoren `?` und `:` . Diese Operatoren ermöglichen die Zuweisung von Dreiergruppen von Anweisungen, in denen der erste Ausdruck zur bedingten Auswertung eines der anderen beiden dient. Beispielsweise könnte die folgende D-Anweisung verwendet werden, um eine Variable `x` je nach dem Wert von `i` auf eine von zwei Zeichenketten zu setzen:

```
x = i == 0 ? "zero" : "non-zero";
```

In diesem Beispiel wird zunächst der Ausdruck `i == 0` ausgewertet, um festzustellen, ob er wahr oder falsch ist. Ist der erste Ausdruck wahr, wird der zweite Ausdruck ausgewertet und der Ausdruck `?:` gibt seinen Wert zurück. Ist der erste Ausdruck falsch, wird der dritte Ausdruck ausgewertet und der Ausdruck `?:` gibt seinen Wert zurück.

Wie bei allen D-Operatoren können Sie in einem Ausdruck mehrere `?:`-Operatoren verwenden und so komplexere Ausdrücke generieren. So würde beispielsweise der folgende Ausdruck die eines der Zeichen 0-9, a-z oder A-Z enthaltende `char`-Variable `c` nehmen und den Wert dieses Zeichens zurückgeben, wenn es als Ziffer in einer hexadezimalen (Grundzahl 16) Zahl interpretiert wird:

```
hexval = (c >= '0' && c <= '9') ? c - '0' :
    (c >= 'a' && c <= 'z') ? c + 10 - 'a' : c + 10 - 'A';
```

Der erste mit `?:` verwendete Ausdruck muss ein Zeiger auf eine Ganzzahl sein, damit er auf seinen Wahrheitswert hin untersucht werden kann. Beim zweiten und dritten Ausdruck kann es sich um einen beliebigen kompatiblen Typ handeln. Die Konstruktion eines bedingten

Ausdrucks ist nicht möglich, wenn beispielsweise ein Pfad eine Zeichenkette und ein anderer Pfad eine Ganzzahl zurückgibt. Außerdem dürfen der zweite und der dritte Ausdruck keine Ablaufverfolgungsfunktion wie `trace()` oder `printf()` aufrufen. Zur bedingten Ablaufverfolgung von Daten greifen Sie, wie in Kapitel 1, „Einführung“ erläutert, stattdessen auf Prädikate zurück.

Typumwandlungen

Bei der Konstruktion von Ausdrücken mit Operanden unterschiedlicher, aber kompatibler Typen werden Typumwandlungen zur Bestimmung des Typs des resultierenden Ausdrucks vorgenommen. Die D-Regeln für Typumwandlungen stimmen mit den Regeln für die arithmetische Konvertierung von Ganzzahlen in ANSI-C überein, die auch als *normale arithmetische Umwandlungen* bezeichnet werden.

Die Umwandlungsregeln lassen sich auf einfache Weise wie folgt beschreiben: Jeder Integer-Typ hat eine Priorität nach der Rangfolge `char, short, int, long, long long`. Dabei hat ein vorzeichenloser Typ gegenüber seiner vorzeichenbehafteten Entsprechung eine höhere, gegenüber dem in der Rangfolge nächsten Integer-Typ aber eine niedrigere Priorität. Wenn Sie einen Ausdruck mit zwei ganzzahligen Operanden wie `x + y` formulieren und die Operanden unterschiedliche Integer-Typen aufweisen, wird dem Ergebnis der Typ des Operanden mit der höchsten Priorität zugewiesen.

Sollte eine Umwandlung erforderlich sein, wird der in der Rangfolge niedrigere Operand zunächst auf den nächst höheren Typ *erweitert.* Durch die Typerweiterung wird der Wert des Operanden nicht wirklich geändert: Es erfolgt lediglich eine Erweiterung seiner Wertemenge im Einklang mit dem Vorzeichen des Werts. Bei einer Typerweiterung eines vorzeichenlosen Operanden werden die nicht verwendeten höherwertigen Bits in der resultierenden Ganzzahl mit Nullen aufgefüllt. Bei einer Typerweiterung eines vorzeichenbehafteten Operanden werden die nicht verwendeten höherwertigen Bits durch eine Vorzeichenerweiterung angefüllt. Wenn ein vorzeichenbehafteter Typ in einen vorzeichenlosen umgewandelt wird, erfolgt zunächst eine Vorzeichenerweiterung des vorzeichenbehafteten Typs und anschließend erhält dieser den neuen, durch die Umwandlung bestimmten vorzeichenlosen Typ.

Integer- und andere Typen können auch explizit von einem Typ in einen anderen umgewandelt werden. Die explizite Typumwandlung wird auch als *Cast* bezeichnet. In D ist die explizite Typumwandlung von Zeigern und Ganzzahlen in beliebige Integer- oder Zeigertypen, nicht aber in andere Typen möglich. Die Regeln für das Casting und die Typerweiterung von Zeichenketten und Zeichenvektoren werden in Kapitel 6, „Zeichenketten“ besprochen. Eine explizite Typumwandlung einer Ganzzahl oder eines Zeigers erfolgt anhand eines Ausdrucks wie diesem:

```
y = (int)x;
```

wobei der Zieltyp eingeklammert ist und als Präfix für den Ausgangsausdruck dient. Die explizite Typumwandlung von Ganzzahlen in höherwertige Typen erfolgt über eine

Typerweiterung. Zur expliziten Typumwandlung von Ganzzahlen in niederwertige Typen werden die überschüssigen höherwertigen Bits in der Ganzzahl mit Nullen aufgefüllt.

Da D keine Gleitkomma-Arithmetik zulässt, ist auch keine implizite oder explizite Umwandlung von Gleitkomma-Operanden erlaubt und sind keine Regeln für die implizite Gleitkomma-Typumwandlung definiert.

Rangfolge

Die D-Regeln für die Operatorrangfolge (auch Präzedenz oder Priorität) und Assoziativität sind in der folgenden Tabelle dargestellt. Diese Regeln sind recht komplex, aber erforderlich für die Gewährleistung einer genauen Kompatibilität mit den ANSI-C-Regeln für die Operatorrangfolge. Die Einträge sind nach absteigender Rangfolge sortiert.

TABELLE 2–11 Operatorrangfolge und Assoziativität in D

Operatoren	Assoziativität
`() [] -> .`	rechtsassoziativ
`! ~ ++ -- + - * &` (*Typ*) `sizeof stringof offsetof xlate`	linksassoziativ
`* / %`	rechtsassoziativ
`+ -`	rechtsassoziativ
`<< >>`	rechtsassoziativ
`< <= > >=`	rechtsassoziativ
`== !=`	rechtsassoziativ
`&`	rechtsassoziativ
`^`	rechtsassoziativ
`|`	rechtsassoziativ
`&&`	rechtsassoziativ
`^^`	rechtsassoziativ
`||`	rechtsassoziativ
`?:`	linksassoziativ
`= += -= *= /= %= &= ^= |= <<= >>=`	linksassoziativ
`,`	rechtsassoziativ

Einige der Operatoren in dieser Tabelle sind uns bisher noch nicht begegnet. Sie werden in späteren Kapiteln besprochen:

`sizeof`	Berechnet die Größe eines Objekts (Kapitel 7, „Strukturen und Unionen“)
`offsetof`	Berechnet den Versatz einer Typkomponente (Kapitel 7, „Strukturen und Unionen“)
`stringof`	Wandelt den Operanden in eine Zeichenkette um (Kapitel 6, „Zeichenketten“)
`xlate`	Übersetzt einen Datentyp (Kapitel 40, „Übersetzer“)
unäres `&`	Berechnet die Adresse eines Objekts (Kapitel 5, „Zeiger und Vektoren“)
unäres `*`	Dereferenziert einen Zeiger auf ein Objekt (Kapitel 5, „Zeiger und Vektoren“)
`->` und `.`	Greift auf die Komponente einer Struktur bzw. einer Union zu(Kapitel 7, „Strukturen und Unionen“)

Der in der Tabelle aufgeführte Komma-Operator (`,`) dient zur Gewährleistung der Kompatibilität mit dem Komma-Operator in ANSI-C, mit dem eine Reihe von Ausdrücken rechtsgerichtet ausgewertet und der Wert des am weiten rechts stehenden Ausdrucks zurückgegeben werden kann. Dieser Operator wird ausschließlich aus Kompatibilitätsgründen bereitgestellt und sollte nicht verwendet werden.

Der Eintrag `()` in der Tabelle der Operatorrangfolge stellt einen Funktionsaufruf dar. Beispiele für Aufrufe von Funktionen wie `printf()` oder `trace()` finden Sie in Kapitel 1, „Einführung“. Kommata werden in D auch zum Auflisten von Argumenten für Funktionen und zum Erstellen von Schlüsseln für assoziative Vektoren verwendet. Dieses Komma ist nicht mit dem Komma-Operator identisch und garantiert *keine* rechtsgerichtete Auswertungsreihenfolge. Der D-Compiler bietet keine Garantie in Bezug auf die Auswertungsreihenfolge der Argumente einer Funktion oder der Schlüssel eines assoziativen Vektors. Bei der Verwendung von Ausdrücken mit interagierenden Nebeneffekten, wie beispielsweise des Ausdruckspaars `i` und `i++`, ist in diesem Kontext Vorsicht geboten.

Der Eintrag `[]` in der Tabelle der Operatorrangfolge stellt eine Referenz auf einen Vektor oder einen assoziativen Vektor dar. Beispiele für assoziative Vektoren finden Sie in Kapitel 1, „Einführung“. Eine Sonderform des assoziativen Vektors, das *Aggregat*, wird in Kapitel 9, „Aggregate“ beschrieben. In Kapitel 5, „Zeiger und Vektoren“ erfahren Sie, wie der Operator `[]` auch zum Indizieren von C-Vektoren fester Größe eingesetzt werden kann.

◆ ◆ ◆ KAPITEL 3

Variablen

D bietet zwei einfache Variablentypen, die Sie in Ihren Tracing-Programmen verwenden können: skalare Variablen und assoziative Vektoren. In den Beispielen in·Kapitel 1 demonstrierten wir bereits kurz die Verwendung dieser Variablen. In diesem Kapitel werden die Regeln zur Verwendung von D-Variablen und die ·Zuweisung von Variablen zu verschiedenen Geltungsbereichen näher erläutert. Eine Sonderform der Vektorvariable, das *Aggregat*, wird in Kapitel 9, „Aggregate“ beschrieben.

Skalare Variablen

Mit skalaren Variablen werden einzelne Datenobjekte fester Größe wie zum Beispiel Ganzzahlen und Zeiger dargestellt. Skalare Variablen können auch Objekte fester Größe aufnehmen, die sich aus einem oder mehreren Grund- oder Verbundtypen zusammensetzen. D bietet die Möglichkeit, sowohl Vektoren von Objekten als auch zusammengesetzte Strukturen zu erstellen. DTrace stellt auch Zeichenketten als skalare Variablen fester Größe dar, indem sie die Möglichkeit erhalten, bis zu einer vordefinierten maximalen Länge anzuwachsen. Wie Sie auf die Zeichenkettenlänge in Ihren D-Programmen Einfluss nehmen, wird in Kapitel 6, „Zeichenketten“ näher besprochen.

Skalare Variablen werden automatisch erzeugt, wenn Sie einem zuvor nicht definierten Bezeichner im D-Programm zum ersten Mal einen Wert zuweisen. Um beispielsweise eine skalare Variable `x` des Typs `int` zu erzeugen, brauchen Sie ihr lediglich in einer beliebigen Prüfpunktklausel einen Wert des Typs `int` zuzuweisen:

```
BEGIN
{
    x = 123;
}
```

Auf diese Weise erzeugte skalare Variablen sind *globaler* Natur: Ihr Name und ihre Datenspeicherposition werden einmal definiert und sind in jeder Klausel des D-Programms

sichtbar. Mit jeder Referenzierung des Bezeichners x verweisen Sie auf eine bestimmte Speicherposition, die dieser Variable zugewiesen wurde.

Im Gegensatz zu ANSI-C erfordert D keine explizite Variablendeklaration. Wenn Sie eine globale Variable dennoch deklarieren möchten, um ihr vor Gebrauch explizit einen Namen und Typ zuzuweisen, können Sie die Deklaration, wie in nachfolgendem Beispiel dargestellt, außerhalb der Prüfpunktklauseln in das Programm einfügen. Explizite Variablendeklarationen sind in den meisten D-Programmen nicht erforderlich, können sich aber dann als hilfreich erweisen, wenn Sie es vorziehen, die Variablentypen genau unter Kontrolle zu halten oder Ihre Programm mit Deklarationen und Kommentaren beginnen zu lassen, die Ihre Programmvariablen und ihre Bedeutung dokumentieren.

```
int x; /* declare an integer x for later use */

BEGIN
{
    x = 123;
    ...
}
```

Im Gegensatz zu ANSI-C-Deklarationen dürfen Variablendeklarationen in D keine Anfangswerte zuweisen. Für die Zuweisung von Anfangswerten müssen Sie eine `BEGIN`-Prüfpunktklausel verwenden. Der gesamte Speicher der globalen Variable wird von DTrace mit Nullen aufgefüllt, bevor Sie die Variable zum ersten Mal referenzieren.

In der Definition der Programmiersprache D besteht keine Beschränkungen für die Größe und Anzahl von D-Variablen. Es sind jedoch Beschränkungen durch die DTrace-Implementierung und die auf dem System verfügbare Speicherkapazität gesetzt. Wenn Sie das Programm kompilieren, werden vom D-Compiler alle anwendbaren Beschränkungen in Kraft gesetzt. In Kapitel 16, „Optionen und Tunables “ erfahren Sie, wie Sie Optionen in Verbindung mit den Programmbeschränkungen abstimmen können.

Assoziative Vektoren

Assoziative Vektoren dienen zum Darstellen von Sammlungen von Datenelementen, die durch Angabe eines Namens, des *Schlüssels*, abgerufen werden können. In D werden die Schlüssel assoziativer Vektoren in Form von Listen skalarer Ausdruckswerte namens *Tupel* gebildet. Das Vektortupel selbst können Sie sich als eine Parameterliste für eine Funktion vorstellen, die zur Ermittlung des entsprechenden Vektorwerts aufgerufen wird, wenn Sie den Vektor referenzieren. Jeder assoziative Vektor in D besitzt eine feststehende *Schlüsselsignatur*, die aus einer festen Anzahl von Tupelelementen besteht. Dabei ist jedem Element ein fester Typ zugeordnet. Sie können für jeden Vektor in einem D-Programm unterschiedliche Schlüsselsignaturen definieren.

Assoziative Vektoren unterscheiden sich von normalen Vektoren mit fester Größe dadurch, dass sie keine vordefinierte Mengenbeschränkung für Elemente haben, die Elemente durch ein

beliebiges Tupel indiziert werden können, anstatt dass nur Ganzzahlen als Schlüssel eingesetzt werden, und schließlich dadurch, dass die Elemente nicht an vorreservierten, aufeinander folgenden Speicherpositionen gespeichert werden. Assoziative Vektoren sind dann hilfreich, wenn Sie in einem Programm in C, C++ oder Java™ eine Hash-Tabelle oder andere einfache Dictionary-Datenstrukturen einsetzen würden. Mit assoziativen Vektoren können Sie ein dynamisches Protokoll der innerhalb des D-Programms erfassten Ereignisse und Statusinformationen erstellen, auf dessen Grundlage sich komplexere Kontrollstrukturen bilden lassen.

Zum Definieren eines assoziativen Vektors schreiben Sie einen Zuweisungsausdruck der Form:

Name `[` *Schlüssel* `]` = *Ausdruck* `;`

dabei ist *Name* ein beliebiger, gültiger D-Bezeichner und *Schlüssel* ist eine Liste eines oder mehrerer durch Komma getrennter Ausdrücke. Beispielsweise definiert die folgende Anweisung einen assoziativen Vektor `a` mit der Schlüsselsignatur `[ int, string ]` und speichert den ganzzahligen Wert 456 an einer nach dem Tupel benannten Position `[ 123, "hello" ]`:

```
a[123, "hello"] = 456;
```

Auch der Objekttyp ist für sämtliche Elemente in einem Vektor festgelegt. Da `a` zuerst die Ganzzahl 456 zugewiesen wurde, erhält auch jeder weitere in diesem Vektor gespeicherte Wert den Typ `int`. Zum Ändern der Elemente in assoziativen Vektoren stehen Ihnen alle in Kapitel 2 definierten Zuweisungsoperatoren zur Verfügung. Dabei sind die für die einzelnen Operatoren genannten Operandenregeln zu beachten. Bei unverträglichen Zuweisungsversuchen gibt der D-Compiler eine entsprechende Fehlermeldung aus. Für die Schlüssel oder Werte assoziativer Vektoren können alle Typen verwendet werden, die auch für skalare Variablen zulässig sind. Es ist nicht möglich, einen assoziativen Vektor als Schlüssel oder Wert in einen anderen assoziativen Vektor zu schachteln.

Ein assoziativer Vektor kann mit jedem mit der Vektorschlüsselsignatur verträglichen Tupel referenziert werden. Die Regeln für die Tupelkompatibilität stimmen mit jenen für Funktionsaufrufe und Variablenzuweisungen überein: Das Tupel muss dieselbe Länge haben, und jeder Typ in der Liste der tatsächlichen Parameter muss mit dem entsprechenden Typ in der formalen Schlüsselsignatur verträglich sein. Wenn zum Beispiel ein assoziativer Vektor `x` wie folgt definiert wird:

```
x[123ull] = 0;
```

dann erhält die Schlüsselsignatur den Typ `unsigned long long` und die Werte nehmen den Typ `int` an. Dieser Vektor kann auch mit dem Ausdruck `x['a']` referenziert werden, da das aus der Zeichenkonstante `'a'` bestehende Tupel des Typs `int` und der Länge 1 gemäß den unter·„Typumwandlungen“ auf Seite 60 beschriebenen arithmetischen Umwandlungsregeln mit der Schlüsselsignatur `unsigned long long` verträglich ist.

Wenn Sie einen assoziativen Vektor in D vor der Verwendung explizit deklarieren müssen, können Sie in den Programmquellcode die Deklaration des Vektornamens und der Schlüsselsignatur außerhalb der Prüfpunktklauseln einfügen.

```
int x[unsigned long long, char];

BEGIN
{
    x[123ull, 'a'] = 456;
}
```

Nach der Definition eines assoziativen Vektors sind Referenzen auf ein beliebiges Tupel mit verträglicher Schlüsselsignatur möglich, selbst dann, wenn noch keine Zuweisung für das betreffende Tupel erfolgt ist. Der Zugriff auf ein Element eines assoziativen Vektors ohne Zuweisung gibt nach Definition ein mit Nullen angefülltes Objekt zurück. Eine Folge dieser Definition ist, dass für das Element des assoziativen Vektors erst nach Zuweisung eines Werts ungleich Null Speicher reserviert wird. Umgekehrt hebt DTrace die Speicherzuweisung auf, wenn einem Element eines assoziativen Vektors der Wert Null zugewiesen wird. Dieses Verhalten ist deshalb wichtig, weil der dynamische Variablenbereich, aus dem die Elemente assoziativer Vektoren zugewiesen werden, endlich ist. Sollte dieser Bereich bei einem Zuweisungsversuch bereits erschöpft sein, schlägt die Zuweisung mit einer Fehlermeldung über eine dynamische Variablenauslassung fehl. Weisen Sie Elementen assoziativer Vektoren, die nicht mehr benötigt werden, immer den Wert Null zu. Weitere Verfahren zur Eliminierung der dynamischen Übergabe von Variablen finden Sie in Kapitel 16, „Optionen und Tunables “.

Thread-lokale Variablen

DTrace bietet die Möglichkeit, im Gegensatz zu den zuvor in diesem Kapitel besprochenen globalen Variablen auch Variablen mit einem Speicherbereich im jeweiligen Betriebssystem-Thread (lokal) zu deklarieren. Thread-lokale Variablen erweisen sich als nützlich, wenn Sie einen Prüfpunkt aktivieren und jeden den Prüfpunkt auslösenden Thread mit einem Etikett oder auf andere Weise markieren möchten. Ein solches Programm zu schreiben, ist in D eine einfache Angelegenheit, denn thread-lokale Variablen haben im D-Code einen gemeinsamen Namen, beziehen sich aber auf unterschiedliche Datenspeicherbereiche der verschiedenen Threads. Thread-lokale Variablen werden durch Anwendung des Operators `->` auf den speziellen Bezeichner `self` referenziert:

```
syscall::read:entry
{
    self->read = 1;
}
```

In diesem D-Beispielfragment wird der Prüfpunkt am `read(2)`-Systemaufruf aktiviert und jedem Thread, der den Prüfpunkt auslöst, wird die thread-lokale Variable `read` zugewiesen.

Ebenso wie globale Variablen werden auch thread-lokale Variablen bei ihrer ersten Zuweisung automatisch erzeugt und nehmen den auf der rechten Seite der ersten Zuweisungsanweisung verwendeten Typ an (in diesem Beispiel `int`).

Mit jeder Referenzierung der Variable `self->read` in Ihrem D-Programm wird das Datenobjekt referenziert, das dem Betriebssystem-Thread angehört, der zum Zeitpunkt der Auslösung des entsprechenden DTrace-Prüfpunkts ausgeführt wurde. Eine thread-lokale Variable kann man sich als einen assoziativen Vektor vorstellen, der durch ein die Identität des Threads im System beschreibendes Tupel implizit indiziert wird. Threads haben über die gesamte Lebensdauer des Systems eine eindeutige Identität: Wenn der Thread beendet wird und aufgrund derselben Betriebssystem-Datenstruktur ein weiterer Thread erzeugt wird, erhält dieser in DTrace *nicht* dieselbe thread-lokale Speicheridentität.

Nachdem Sie eine thread-lokale Variable definiert haben, können Sie diese für jeden beliebigen Thread im System referenzieren, selbst wenn die betreffende Variable für einen bestimmten Thread noch keine Zuweisung erhalten hat. Der Datenspeicherbereich für eine Kopie der thread-lokalen Variable im Thread wird gemäß der Definition mit Nullen angefüllt. Ebenso wie bei den Elementen assoziativer Vektoren erfolgt die Speicherreservierung für eine thread-lokale Variable erst, wenn dieser ein Wert ungleich Null zugewiesen wird. Ebenso analog zu den Elementen assoziativer Vektoren bewirkt die Zuweisung mit Null einer thread-lokalen Variable, dass DTrace die Zuweisung des Speicherbereichs aufhebt. Weisen Sie thread-lokalen Variablen, die nicht mehr benötigt werden, immer den Wert Null zu. In Kapitel 16, „Optionen und Tunables “ sind weitere Techniken zur Feinabstimmung des dynamischen Variablenbereichs beschrieben, aus dem thread-lokale Variablen allokiert werden.

Sie können in einem D-Programm thread-lokale Variablen jedes beliebigen Typs, einschließlich assoziativer Vektoren, definieren. Sehen Sie hier einige Beispieldefinitionen für thread-lokale Variablen:

```
self->x = 123;              /* integer value */
self->s = "hello";             /* string value */
self->a[123, 'a'] = 456;    /* associative array */
```

Wie jede D-Variable müssen auch thread-lokale Variablen vor ihrer Verwendung nicht explizit deklariert werden. Wenn Sie dies trotzdem wünschen, bauen Sie die Deklaration außerhalb der Programmklauseln ein und stellen Sie ihnen das Schlüsselwort `self` voran:

```
self int x;    /* declare int x as a thread-local variable */

syscall::read:entry
{
    self->x = 123;
}
```

Thread-lokale Variablen werden von globalen Variablen getrennt in separaten Namensräumen geführt. Folglich können die Namen mehrmals verwendet werden. Denken Sie daran, dass es

sich bei x und self->x nicht um dieselbe Variable handelt, wenn Sie Namen in Ihrem Programm mehrmals verwenden! Das folgende Beispiel verdeutlicht die Verwendung von thread-lokalen Variablen. Geben Sie das folgende Programm in einen Texteditor ein und speichern Sie es unter dem Namen rtime.d:

BEISPIEL 3–1 rtime.d: Berechnen der in read(2) abgelaufenen Zeit

```
syscall::read:entry
{
    self->t = timestamp;
}

syscall::read:return
/self->t != 0/
{
    printf("%d/%d spent %d nsecs in read(2)\n",
        pid, tid, timestamp - self->t);

    /*
     * We're done with this thread-local variable; assign zero to it to
     * allow the DTrace runtime to reclaim the underlying storage.
     */
    self->t = 0;
}
```

Wechseln Sie nun zu Ihrer Shell und starten Sie die Programmausführung. Nach einigen Sekunden sollte die Ausgabe beginnen. Wenn keine Ausgabe erscheint, geben Sie den ein oder anderen Befehl ein.

```
# dtrace -q -s rtime.d
100480/1 spent 11898 nsecs in read(2)
100441/1 spent 6742 nsecs in read(2)
100480/1 spent 4619 nsecs in read(2)
100452/1 spent 19560 nsecs in read(2)
100452/1 spent 3648 nsecs in read(2)
100441/1 spent 6645 nsecs in read(2)
100452/1 spent 5168 nsecs in read(2)
100452/1 spent 20329 nsecs in read(2)
100452/1 spent 3596 nsecs in read(2)
...
^C
#
```

In rtime.d wird mit der thread-lokalen Variable t beim Eintritt in read(2) durch einen beliebigen Thread eine Zeitmarke erfasst. In der Rückkehrklausel gibt das Programm dann die in read(2); verbrachte Zeit aus, die es ermittelt, indem es self->t von der aktuellen Zeitmarke subtrahiert. Die integrierten D-Variablen pid und tid melden die Prozess-ID und Thread-ID

des Threads, der read(2) durchführt. Da self->t nach der Ausgabe dieser Information nicht mehr benötigt wird, wird ihr anschließend der Wert 0 zugewiesen, um DTrace die Möglichkeit zu geben, den Speicherbereich von t für den aktuellen Thread anderweitig zu verwenden.

Normalerweise sehen Sie eine etliche Zeilen lange Ausgabe, auch ohne Befehle eingeben zu müssen, da read(2) im Hintergrund ständig von Serverprozessen und Dämonen ausgeführt wird. Ändern Sie die zweite Klausel von rtime.d ab, indem Sie die Variable execname einsetzen, um zusätzlich den Namen des read(2) ausführenden Prozesses zu erfahren:

```
printf("%s/%d spent %d nsecs in read(2)\n",
    execname, tid, timestamp - self->t);
```

Wenn Sie auf einen Prozess stoßen, der Sie besonders interessiert, fügen Sie ein Prädikat ein, um seinem read(2)-Verhalten genauer auf den Grund zu gehen:

```
syscall::read:entry
/execname == "Xsun"/
{
    self->t = timestamp;
}
```

Klausel-lokale Variablen

Es besteht auch die Möglichkeit, D-Variablen zu deklarieren, deren Speicherbereich für jede D-Programmklausel wieder verwendet wird. Klausel-lokale Variablen sind mit automatischen Variablen in C-, C++- oder Java-Programmen vergleichbar, die während jedes Aufrufs einer Funktion aktiv sind. Wie alle anderen D-Programmvariablen werden auch klausel-lokale Variablen bei ihrer ersten Zuweisung erzeugt. Diese Variablen werden durch Anwendung des Operators -> auf den speziellen Bezeichner this referenziert und zugewiesen:

```
BEGIN
{
    this->secs = timestamp / 1000000000;
    ...
}
```

Wenn Sie eine klausel-lokale Variable vor ihrer Verwendung explizit deklarieren möchten, greifen Sie hierzu auf das Schlüsselwort this zurück:

```
this int x;   /* an integer clause-local variable */
this char c;  /* a character clause-local variable */

BEGIN
{
    this->x = 123;
```

```
    this->c = 'D';
}
```

Klausel-lokale Variablen sind nur während der Lebensdauer der jeweiligen Prüfpunktklausel aktiv. Wenn DTrace die Aktionen der Klauseln für einen bestimmten Prüfpunkt durchgeführt hat, wird der Speicherbereich aller klausel-lokalen Variablen zurückgefordert und für die nächste Klausel verwendet. Deshalb werden klausel-lokale Variablen im Gegensatz zu allen anderen D-Variablen anfänglich nicht mit Nullen angefüllt. Wenn ein Programm mehrere Klauseln für denselben Prüfpunkt enthält, bleiben alle klausel-lokalen Variablen intakt, solange die Klauseln ausgeführt werden. Das folgende Beispiel verdeutlicht dieses Prinzip:

BEISPIEL 3–2 `clause.d`: Klausel-lokale Variablen

```
int me;            /* an integer global variable */
this int foo;        /* an integer clause-local variable */

tick-1sec
{
    /*
     * Set foo to be 10 if and only if this is the first clause executed.
     */
    this->foo = (me % 3 == 0) ? 10 : this->foo;
    printf("Clause 1 is number %d; foo is %d\n", me++ % 3, this->foo++);
}

tick-1sec
{
    /*
     * Set foo to be 20 if and only if this is the first clause executed.
     */
    this->foo = (me % 3 == 0) ? 20 : this->foo;
    printf("Clause 2 is number %d; foo is %d\n", me++ % 3, this->foo++);
}

tick-1sec
{
    /*
     * Set foo to be 30 if and only if this is the first clause executed.
     */
    this->foo = (me % 3 == 0) ? 30 : this->foo;
    printf("Clause 3 is number %d; foo is %d\n", me++ % 3, this->foo++);
}
```

Da die Klauseln *immer* in der Programmreihenfolge ausgeführt werden und klausel-lokale Variablen in unterschiedlichen Klauseln zur Aktivierung desselben Prüfpunkts bestehen, ergibt die Ausführung des obigen Programms stets dieselbe Ausgabe:

```
# dtrace -q -s clause.d
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
Clause 1 is number 0; foo is 10
Clause 2 is number 1; foo is 11
Clause 3 is number 2; foo is 12
^C
```

Während klausel-lokale Variablen über die Klauseln hinweg beständig sind, die denselben Prüfpunkt aktivieren, sind deren Werte in der als Erste für einen Prüfpunkt ausgeführten Klausel nicht definiert. Denken Sie daran, jeder klausel-lokalen Variable vor ihrer Verwendung einen geeigneten Wert zuzuweisen. Anderenfalls bringt das Programm möglicherweise unerwartete Resultate.

Klausel-lokale Variablen können mit jedem skalaren Variablentyp definiert werden. Eine Definition von assoziativen Vektoren mit klausel-lokalem Gültigkeitsbereich ist jedoch nicht möglich. Der Gültigkeitsbereich klausel-lokaler Variablen bezieht sich nur auf die entsprechenden Variablendaten, nicht aber auf den für die Variable definierten Namen oder ihre Typidentität. Die Namens- und Typensignatur einer definierten klausel-lokalen Variable kann in jeder nachfolgenden D-Programmklausel verwendet werden. Es besteht keine Garantie, dass der Speicherbereich über die verschiedenen Klauseln hinweg identisch ist.

Klausel-lokale Variablen können zum Ansammeln von Zwischenergebnissen bei Berechnungen oder als temporäre Kopien anderer Variablen eingesetzt werden. Der Zugriff auf eine klausel-lokale Variable läuft wesentlich schneller ab als der Zugriff auf einen assoziativen Vektor. Wenn Sie also den Wert eines assoziativen Vektors in derselben D-Programmklausel mehrmals referenzieren müssen, erweist es sich als effizienter, den Wert zuerst in eine klausel-lokale Variable zu kopieren und die lokale Variable dann wiederholt zu referenzieren.

Integrierte Variablen

Die folgende Tabelle enthält eine vollständige Liste der in D integrierten Variablen. Dabei handelt es sich in allen Fällen um skalare, globale Variablen. Derzeit sind in D weder thread-lokale oder klausel-lokale Variablen noch integrierte assoziative Vektoren definiert.

TABELLE 3–1 Integrierte Variablen in DTrace

Typ und Name	Beschreibung
`int64_t arg0, ..., arg9`	Die ersten zehn Eingangsargumente für einen Prüfpunkt, dargestellt als unbearbeitete 64-Bit-Ganzzahlen. Werden dem aktuellen Prüfpunkt weniger als zehn Argumente übergeben, dann werden die übrigen Variablen auf Null zurückgesetzt.
`args[]`	Ggf. die für den aktuellen Prüfpunkt eingegebenen Argumente. Auf den Vektor `args[]` wird über einen Ganzzahlenindex zugegriffen, doch die einzelnen Elemente sind als der dem jeweiligen Prüfpunktargument entsprechende Typ definiert. Wenn beispielsweise `args[]` durch einen Prüfpunkt für den Systemaufruf `read(2)` referenziert wird, nimmt `args[0]` den Typ `int` an, `args[1]` den Typ `void *` und `args[2]` den Typ `size_t`.
`uintptr_t caller`	Die Speicheradresse des aktuellen Threads im Programmschrittzähler kurz vor Eintritt in den aktuellen Prüfpunkt.
`chipid_t chip`	Die CPU-Chipkennung des aktuellen physischen Chips. Weitere Informationen finden Sie in Kapitel 26, „Der Provider `sched`“.
`processorid_t cpu`	Die CPU-Kennung der aktuellen CPU. Weitere Informationen finden Sie in Kapitel 26, „Der Provider `sched`“.
`cpuinfo_t *curcpu`	Die CPU-Informationen der aktuellen CPU. Weitere Informationen finden Sie in Kapitel 26, „Der Provider `sched`“.
`lwpsinfo_t *curlwpsinfo`	Der LWP-Status des leichtgewichtigen Prozesses (LWP), der für den aktuellen Thread steht. Diese Struktur ist in der Manpage `proc(4)` ausführlicher beschrieben.
`psinfo_t *curpsinfo`	Der Prozessstatus des Prozesses, der für den aktuellen Thread steht. Diese Struktur ist in der Manpage `proc(4)` ausführlicher beschrieben.
`kthread_t *curthread`	Die Adresse der im Betriebssystemkernel internen Datenstruktur für den aktuellen Thread `kthread_t`. Der Thread `kthread_t` ist in `<sys/thread.h>` definiert. Weitere Informationen zu dieser Variable und anderen Betriebssystemdatenstrukturen finden Sie in *Solaris Internals*.

TABELLE 3–1 Integrierte Variablen in DTrace *(Fortsetzung)*

Typ und Name	Beschreibung
`string cwd`	Name des aktuellen Arbeitsverzeichnisses des Prozesses, der für den aktuellen Thread steht.
`uint_t epid`	Die EPID (ID des aktivierten Prüfpunkts) für den aktuellen Prüfpunkt. Diese Ganzzahl ist eine eindeutige Kennung eines bestimmten Prüfpunkts, der mit einem spezifischen Prädikat und Aktionssatz aktiviert wurde.
`int errno`	Der von dem zuletzt durch diesen Thread ausgeführten Systemaufruf zurückgegebene Fehlerwert.
`string execname`	Der für die Ausführung des aktuellen Prozesses an `exec(2)` übergebene Name.
`gid_t gid`	Die reale Gruppen-ID des aktuellen Prozesses.
`uint_t id`	Die Prüfpunkt-ID für den aktuellen Prüfpunkt. Dabei handelt es sich um die systemweite, eindeutige Kennung des Prüfpunkts, wie sie von DTrace veröffentlicht und in der Ausgabe von `dtrace -l` angegeben wird.
`uint_t ipl`	Die Interrupt-Priorität (IPL) auf der aktuellen CPU zum Zeitpunkt der Auslösung des Prüfpunkts. Weitere Informationen zu Interrupt-Prioritäten und zur Interrupt-Behandlung im Solaris-Betriebssystemkernel finden Sie in *Solaris Internals*.
`lgrp_id_t lgrp`	Die Latenzgruppen-ID für die Latenzgruppe, der die aktuelle CPU angehört. Weitere Informationen finden Sie in Kapitel 26, „Der Provider `sched`“.
`pid_t pid`	Die Prozess-ID des aktuellen Prozesses.
`pid_t ppid`	Die Prozess-ID des dem aktuellen Prozess übergeordneten Prozesses.
`string probefunc`	Der Funktionsname als Bestandteil der aktuellen Prüfpunktbeschreibung.
`string probemod`	Der Modulname als Bestandteil der aktuellen Prüfpunktbeschreibung.
`string probename`	Der Name als Bestandteil der aktuellen Prüfpunktbeschreibung.

TABELLE 3–1 Integrierte Variablen in DTrace *(Fortsetzung)*

Typ und Name	Beschreibung
`string probeprov`	Der Providername als Bestandteil der aktuellen Prüfpunktbeschreibung.
`psetid_t pset`	Die Prozessorsatz-ID des Prozessorsatzes, der die aktuelle CPU enthält. Weitere Informationen finden Sie in Kapitel 26, „Der Provider `sched`“.
`string root`	Name des Root-Verzeichnisses des Prozesses, der für den aktuellen Thread steht.
`uint_t stackdepth`	Die Stacktiefe des aktuellen Threads zum Zeitpunkt der Prüfpunktauslösung.
`id_t tid`	Die Thread-ID des aktuellen Threads. Bei Threads für Benutzerprozesse stimmt dieser Wert mit dem Ergebnis des Aufrufs von `pthread_self`(3C) überein.
`uint64_t timestamp`	Der aktuelle Wert eines Nanosekundenzählers für Zeitmarken. Dieser zählt (wird erhöht) ab einem beliebigen Punkt in der Vergangenheit und sollte nur für relative Berechnungen eingesetzt werden.
`uid_t uid`	Die reale Benutzer-ID des aktuellen Prozesses.
`uint64_t uregs[]`	Die für den aktuellen Thread gespeicherten Benutzermodus-Registerwerte zum Zeitpunkt der Prüfpunktauslösung. Die Verwendung des `uregs[]`-Vektors wird in Kapitel 33, „Ablaufverfolgung von Benutzerprozessen“ behandelt.
`uint64_t vtimestamp`	Der aktuelle Wert eines Nanosekundenzählers für Zeitmarken, der auf die Ausführungsdauer des aktuellem Threads auf einer CPU minus der in DTrace-Prädikaten und -Aktionen verbrachten Zeit virtualisiert ist. Dieser zählt (wird erhöht) ab einem beliebigen Punkt in der Vergangenheit und sollte nur für relative Zeitberechnungen eingesetzt werden.
`uint64_t walltimestamp`	Die seit 00.00 Uhr UCT am 1. Januar 1970 verstrichene Zeit in Nanosekunden.

In D integrierte Funktionen wie `trace()` werden in Kapitel 10, „Aktionen und Subroutinen“ erläutert.

Externe Variablen

In D kommt das Backquote-Zeichen (oder Accent grave) (‘) als spezieller Bereichsoperator zum Ansprechen von Variablen zum Einsatz, die zwar im Betriebssystem, nicht aber im D-Programm definiert sind. So enthält der Solaris-Kernel beispielsweise eine C-Deklaration eines über das System abstimmbaren Parameters (Tunables) namens `kmem_flags` zum Aktivieren von Debugging-Leistungsmerkmalen für die Speicherzuweisung. Im *Solaris Tunable Parameters Reference Manual* finden Sie weitere Informationen zu `kmem_flags`. Dieses Tunable ist im Kernel-Quellcode wie folgt als eine C-Variable deklariert:

```
int kmem_flags;
```

Um in einem D-Programm auf den Wert dieser Variable zuzugreifen, verwenden Sie die D-Notation:

```
‘kmem_flags
```

DTrace weist jedem Kernelsymbol den im entsprechenden C-Betriebssystemcode für das Symbol verwendeten Typ zu und bietet dadurch einen einfachen quellcodebasierten Zugriff auf die nativen Datenstrukturen des Betriebssystems. Wenn Sie externe Betriebssystemvariablen einsetzen möchten, benötigen Sie Zugriff auf den entsprechenden Betriebssystemquellcode.

Mit dem Zugriff auf externe Variablen über ein D-Programm sprechen Sie die internen Implementierungsinformationen eines anderen Programms an, beispielsweise des Betriebssystemkernels oder dessen Gerätetreiber. Diese Implementierungsinformationen stellen keine stabile Schnittstelle dar, auf die Sie sich verlassen könnten! Mit D-Programmen, die von diesen Informationen abhängen, riskieren Sie, dass diese nach dem nächsten Upgrade der entsprechenden Software nicht mehr funktionsfähig sind. Aus diesem Grund werden externe Variablen in der Regel von Kernel- und Gerätetreiberentwicklern und von Wartungspersonal zum Aufspüren von Leistungs- oder Funktionsproblemen mit DTrace verwendet. Weitere Informationen über die Stabilität von D-Programmen finden Sie in Kapitel 39, „Stabilität“.

Die Namen von Kernelsymbolen werden in einem von den D-Variablen- und Funktionsbezeichnern getrennten Namensraum gehalten, sodass keine Gefahr eines Konflikts zwischen diesen Namen und Ihren D-Variablen besteht. Stößt der D-Compiler auf eine Variable mit vorangestelltem Backquote-Zeichen, durchsucht er der Reihenfolge nach die bekannten Kernelsymbole unter Bezugnahme auf die Liste der geladenen Module nach einer passenden Variablendefinition. Da der Solaris-Kernel dynamisch geladene Module mit separaten Symbol-Namensräumen unterstützt, kann ein Variablenname im aktiven Betriebssystemkernel mehrmals verwendet werden. Diese Namenskonflikte können Sie lösen, indem Sie vor dem Backquote-Zeichen im Symbolnamen den Namen des Kernelmoduls angeben, auf dessen Variable zugegriffen werden soll. Beispielsweise bietet jedes ladbare Kernelmodul naturgemäß die Funktion `_fini(9E)`, sodass Sie für die Bezugnahme auf die Adresse der Funktion `_fini`, die ein Kernelmodul namens `foo` zur Verfügung stellt, Folgendes schreiben würden:

```
foo`_fini
```

Alle beliebigen D-Operatoren außer denjenigen, die Werte ändern, können unter Beachtung der üblichen Regeln für Operandentypen auf externe Variablen angewendet werden. Wenn Sie DTrace starten, lädt der D-Compiler die für die aktiven Kernelmodule zutreffenden Variablennamen ein. Eine Deklaration dieser Variablen ist also nicht erforderlich. Operatoren wie beispielsweise = oder +=, die den Wert einer Variable ändern, dürfen nicht auf externe Variablen angewendet werden. Aus Sicherheitsgründen verhindert DTrace eine potenzielle Schädigung des Status der untersuchten Software.

◆ ◆ ◆ KAPITEL 4

D-Programmstruktur

D-Programme bestehen aus einem Satz die zu aktivierenden Prüfpunkte beschreibender Klauseln und aus Prädikaten und Aktionen, die an diese Prüfpunkte gebunden werden. D-Programme enthalten auch Variablendeklarationen (siehe Kapitel 3, „Variablen“) und Definitionen neuer Datentypen (siehe Kapitel 8, „Typ- und Konstantendefinitionen“). Dieses Kapitel bietet eine formelle Beschreibung der Gesamtstruktur eines D-Programms und der Leistungsmerkmale zur Erstellung von Prüfpunktbeschreibungen, die auf mehrere Prüfpunkte zutreffen. Darüber hinaus wird der Einsatz des C-Preprozessors `cpp` in D-Programmen besprochen.

Prüfpunktklauseln und Deklarationen

Wie mit unseren bisherigen Beispielen gezeigt, besteht die Quelldatei eines D-Programms aus mindestens einer Prüfpunktklausel, mit der Sie die von DTrace zu aktivierende Instrumentation beschreiben. Eine Prüfpunktklausel hat die allgemeine Form:

Prüfpunktbeschreibungen
`/` *Prädikat*
`/ {`
Aktionsanweisungen
`}`

Das Prädikat und die Liste der Aktionsanweisungen können ausgelassen werden. Sämtliche Anweisungen außerhalb von Prüfpunktklauseln werden als *Deklarationen* bezeichnet. Deklarationen dürfen nur außerhalb von Prüfpunktklauseln stehen. Deklarationen sind weder innerhalb der geschweiften Klammern `{ }` noch eingestreut zwischen die o. g. Elemente der Prüfpunktklausel zulässig. Leerstellen dürfen zum Trennen beliebiger D-Programmelemente und zum Einrücken von Aktionsanweisungen verwendet werden.

Mit Deklarationen können Sie entweder D-Variablen und externe C-Symbole deklarieren (siehe Kapitel 3, „Variablen“) oder neue Typen für die Verwendung in D definieren (siehe Kapitel 8, „Typ- und Konstantendefinitionen“). Spezielle D-Compiler-Anweisungen, die so

genannten *Pragmas*, können ebenfalls an jeder beliebigen Stelle in einem D-Programm, einschließlich außerhalb von Prüfpunktklauseln, stehen. D-Pragmas werden in Zeilen mit einleitendem #-Zeichen angegeben. Sie dienen beispielsweise zum Setzen von DTrace-Laufzeitoptionen. Näheres hierzu finden Sie in Kapitel 16, „Optionen und Tunables “.

Prüfpunktbeschreibungen

Jede D-Programmklausel beginnt mit einer Liste von einer oder mehreren Prüfpunktbeschreibungen in der üblichen Form:

Provider:*Modul*:*Funktion*:*Name*

Wenn eines oder mehrere Felder der Prüfpunktbeschreibung ausgelassen wird, werden die angegebenen Felder vom D-Compiler von rechts nach links interpretiert. So würde beispielsweise die Prüfpunktbeschreibung `foo:bar` unabhängig von dem Wert der Provider- und Modulfelder auf einen Prüfpunkt mit der Funktion `foo` und dem Namen `bar` zutreffen. Aus diesem Grund ist eine Prüfpunktbeschreibung tatsächlich eher als ein *Muster* zu betrachten, das zur Bezugnahme auf einen oder mehrere Prüfpunkte anhand ihrer Namen dient.

Es empfiehlt sich, in der D-Prüfpunktbeschreibung alle vier Feldtrenner zu verwenden, sodass Sie auf der linken Seite den gewünschten *Provider* angeben können. Bei Verzicht auf die Angabe eines Providers erhalten Sie möglicherweise unerwartete Resultate, wenn mehrere Provider Prüfpunkte mit demselben Namen veröffentlichen. Ebenso ist es nicht auszuschließen, dass zukünftige Versionen von DTrace neue Provider enthalten, deren Prüfpunkte zufällig mit den von Ihnen nur teilweise angegebenen Prüfpunktbeschreibungen übereinstimmen. Um einen Provider mit all seinen Prüfpunkten anzugeben, lassen Sie die Modul-, Funktions- und Namensfelder leer. So können Sie beispielsweise mit der Beschreibung `syscall:::` jeden von dem DTrace-Provider `syscall` veröffentlichten Prüfpunkt ansprechen.

Prüfpunktbeschreibungen unterstützen auch eine Mustervergleichssyntax wie die in `sh(1)` beschriebene *Globbing*-Syntax. Bevor DTrace einen Prüfpunkt für eine Beschreibung einsetzt, werden alle Beschreibungsfelder nach den Zeichen `*`, `?` und `[` durchsucht. Wenn eines dieser Zeichen in einem Feld der Prüfpunktbeschreibung ohne vorangestelltes `\`-Zeichen vorkommt, wird das Feld als Suchmuster interpretiert. Das Beschreibungsmuster muss mit dem gesamten entsprechenden Feld eines gegebenen Prüfpunkts übereinstimmen. Um einen Prüfpunkt erfolgreich zu identifizieren und zu aktivieren, muss die vollständige Prüfpunktbeschreibung in jedem Feld übereinstimmen. Ein Prüfpunktbeschreibungsfeld, bei dem es sich nicht um ein Suchmuster handelt, muss exakt mit dem entsprechenden Feld des Prüfpunkts übereinstimmen. Leere Beschreibungsfelder treffen auf alle Prüfpunkte zu.

In Suchmustern für Prüfpunktnamen werden die Sonderzeichen in folgender Tabelle erkannt:

TABELLE 4–1 Mustervergleichszeichen für Prüfpunktnamen

Symbol	Beschreibung
*	Trifft auf jede Zeichenkette einschließlich der Null-Zeichenkette zu.
?	Trifft auf jedes Einzelzeichen zu.
[...]	Trifft auf jedes der eingeklammerten Zeichen zu. Ein durch - getrenntes Zeichenpaar trifft auf jedes Zeichen zwischen den beiden Zeichen (einschließlich dieser) zu. Wenn das erste Zeichen nach der öffnenden Klammer [ein ! ist, besteht Übereinstimmung mit allen nicht im Satz enthaltenen Zeichen.
\	Das nächste Zeichen wird ohne Sonderbedeutung in seiner normalen Bedeutung interpretiert.

Mustervergleichszeichen können in beliebigen oder allen vier Feldern einer Prüfpunktbeschreibung verwendet werden. Mit `dtrace -l` können Sie Suchmuster für übereinstimmende Prüfpunkte auch in der Befehlszeile angeben. Beispielsweise listet der Befehl `dtrace -l -f kmem_*` alle DTrace-Prüfpunkte in Funktionen auf, deren Namen mit dem Präfix `kmem_` beginnen.

Wenn Sie dasselbe Prädikat und dieselben Aktionen für mehrere Prüfpunktbeschreibungen oder Beschreibungsmuster angeben möchten, können Sie die Beschreibungen durch Kommata getrennt auflisten. So würde beispielsweise das folgende D-Programm bei jeder Auslösung von Prüfpunkten im Zusammenhang mit dem Eintritt in Systemaufrufe, die die Wörter „`lwp`" oder „`sock`" enthalten, eine Zeitmarke protokollieren:

```
syscall::*lwp*:entry, syscall::*sock*:entry
{
    trace(timestamp);
}
```

Prüfpunkte können in Prüfpunktbeschreibungen auch anhand ihrer ganzzahligen Prüfpunkt-ID angegeben werden. So kann beispielsweise mit der Klausel:

```
12345
{
    trace(timestamp);
}
```

der Prüfpunkt mit der ID 12345, der von `dtrace -l -i 12345` gemeldet wird, aktiviert werden. Fügen Sie in Ihre D-Programme stets vom Menschen lesbare Prüfpunktbeschreibungen ein. Numerische Prüfpunkt-IDs bleiben über das Einladen und Entfernen der DTrace-Provider-Kernelmodule hinweg oder nach einem Systemneustart möglicherweise nicht unverändert bestehen.

Prädikate

Prädikate sind zwischen Schrägstrichen `/ /` stehende Ausdrücke, die zum Zeitpunkt der Prüfpunktauslösung ausgewertet werden, um festzustellen, ob die zugehörigen Aktionen ausgeführt werden sollen. Prädikate sind das wichtigste bedingte Konstrukt für die Bildung komplexerer Kontrollflüsse in einem D-Programm. Sie können für jeden Prüfpunkt den Prädikatteil der Prüfpunktklausel ganz auslassen. In diesem Fall werden die Aktionen bei der Auslösung des Prüfpunkts immer ausgeführt.

Prädikatausdrücke akzeptieren alle bisher beschriebenen D-Operatoren und können sich auf beliebige D-Datenobjekte wie zum Beispiel Variablen und Konstanten beziehen. Der Prädikatausdruck muss einen Wert des Integer- oder Zeigertyps ergeben, sodass er als wahr oder falsch interpretiert werden kann. Wie bei allen D-Ausdrücken wird ein Nullwert als falsch und ein Wert ungleich Null als wahr interpretiert.

Aktionen

Die Beschreibung von Prüfpunktaktionen erfolgt in Form einer Liste durch Strichpunkt (`;`) getrennter Anweisungen in geschweiften Klammern `{ }`. Wenn Sie nur feststellen möchten, ob ein bestimmter Prüfpunkt auf einer bestimmten CPU ausgelöst wurde, ohne Daten zu verfolgen oder zusätzliche Aktionen durchzuführen, können Sie ein leeres geschweiftes Klammernpaar ohne darin enthaltene Anweisungen angeben.

Einsatz des C-Preprozessors

Die Programmiersprache C, in der die Solaris-Systemschnittstellen definiert sind, umfasst einen *Preprozessor*, der einige Anfangsschritte bei der C-Programmkompilierung durchführt. Der C-Preprozessor wird in der Regel zum Definieren von Makroersetzungen, bei denen ein Symbol (Token) in einem C-Programm durch einen vordefinierten Satz anderer Symbole ersetzt wird, oder zum Anlegen von Kopien der Systemdefinitionsdateien verwendet. Wenn Sie den C-Preprozessor in Verbindung mit den D-Programmen verwenden möchten, geben Sie die `dtrace`-Option `-C` an. Diese Option bewirkt, dass `dtrace` zuerst den `cpp(1)`-Preprozessor auf Ihrer Programmquelldatei ausführt und die Ergebnisse dann an den D-Compiler weiterleitet. Eine ausführlichere Beschreibung des C-Preprozessors finden Sie in *Programmieren in C*.

Der D-Compiler lädt die C-Typbeschreibungen für die entsprechende Betriebssystemimplementierung zwar automatisch ein, doch mit dem Preprozessor haben Sie die Möglichkeit, weitere Typdefinitionen wie zum Beispiel Typen aus eigenen C-Programmen einzubinden. Der Preprozessor bietet sich auch für andere Vorgänge wie etwa die Erstellung von Makros an, die sich durch D-Codechunks und andere Programmelemente ersetzen lassen. Wenn Sie den Preprozessor für ein D-Programm verwenden, dürfen Sie nur Dateien mit gültigen D-Deklarationen aufnehmen. Typische C-Definitionsdateien laden nur externe

Deklarationen von Typen und Symbolen, die der D-Compiler fehlerfrei interpretiert. Der D-Compiler kann C-Definitionsdateien mit zusätzlichen Programmelementen wie C-Funktionsquellcode nicht analysieren und generiert in diesen Fällen eine entsprechende Fehlermeldung.

◆ ◆ ◆ KAPITEL 5

Zeiger und Vektoren

Zeiger sind Speicheradressen von Datenobjekten im Betriebssystemkernel oder im Adressbereich eines Benutzerprozesses. D bietet die Fähigkeit, Zeiger zu erstellen und zu manipulieren und sie in Variablen und assoziativen Vektoren zu speichern. Dieses Kapitel befasst sich mit der D-Syntax für Zeiger, Operatoren, die beim Erstellen von Zeigern oder beim Zugriff auf sie angewendet werden können, sowie mit der Beziehung zwischen Zeigern und skalaren Vektoren fester Größe. Darüber hinaus werden Probleme im Zusammenhang mit dem Einsatz von Zeigern in unterschiedlichen Adressbereichen besprochen.

Hinweis – Wenn Sie viel Erfahrung mit dem Programmieren in C oder C++ besitzen, können Sie einen Großteil dieses Kapitels durchaus überfliegen, da die D-Zeigersyntax mit der entsprechenden ANSI-C-Syntax übereinstimmt. Lesen Sie die Abschnitte „Zeiger auf DTrace-Objekte“ auf Seite 90 und „Zeiger und Adressräume“ auf Seite 91, die DTrace-spezifische Merkmale und Aspekte beschreiben.

Zeiger und Adressen

Im Betriebssystem Solaris wird jedem Benutzerprozess mithilfe von *virtuellem Speicher* eine eigene virtuelle Sicht der Speicherressourcen auf dem System bereitgestellt. Eine virtuelle Sicht der Speicherressourcen wird als *Adressraum* bezeichnet. Durch ihn wird ein Bereich von Adresswerten (entweder `[0 ... 0xffffffff]` bei 32-Bit-Adressräumen oder `[0 ... 0xffffffffffffffff]` bei 64-Bit-Adressräumen) bestimmten Übersetzungen zugewiesen, auf deren Grundlage das Betriebssystem und die Hardware die einzelnen virtuellen Adressen in einen entsprechenden Speicherbereich im physischen Speicher konvertieren. In D stellen Zeiger Datenobjekte dar, die einen Ganzzahlwert einer virtuellen Adresse speichern und ihm einen D-Typ zuordnen, der das Format der an der entsprechenden Speicherposition abgelegten Daten beschreibt.

Um eine D-Variable als Zeigertyp zu deklarieren, geben Sie zunächst den Typ der referenzierten Daten an und fügen dann ein Sternchen (`*`) an den Typnamen an, womit Sie zu erkennen geben, dass ein Zeigertyp deklariert werden soll. So wird beispielsweise die Deklaration:

```
int *p;
```

eine globale D-Variable namens `p` deklariert werden, die einen Zeiger auf eine Ganzzahl darstellt. Diese Deklaration bedeutet, dass `p` selbst eine 32- oder 64-Bit-Ganzzahl ist, deren Wert die Adresse einer anderen Ganzzahl irgendwo im Speicher darstellt. Da die kompilierte Form des D-Codes zum Zeitpunkt der Prüfpunktauslösung innerhalb des Betriebssystemkernels selbst ausgeführt wird, beziehen sich Zeiger in D typischerweise auf den Kernel-Adressraum. Mit dem Befehl `isainfo(1) -b` lässt sich ermitteln, wie viele Bits der aktive Betriebssystemkernel auf Zeiger verwendet.

Wenn Sie einen Zeiger auf ein Datenobjekt im Kernel erstellen möchten, können Sie dessen Adresse mit dem Operator `&` berechnen. Nehmen wir beispielsweise das im Betriebssystemkern-Quellcode deklarierte Tunable `int kmem_flags.` Sie könnten die Adresse dieses `int`-Tunables erfassen, indem Sie das Ergebnis der Anwendung des Operators `&` auf den Namen dieses Objekts in D verfolgen:

```
trace(&‘kmem_flags);
```

Mit dem Operator * können Sie das durch den Zeiger angesprochene Objekt referenzieren. Der Operator hat die umgekehrte Wirkung des Operators &. So haben beispielsweise die beiden folgenden D-Codefragmente dieselbe Bedeutung:

```
p = &‘kmem_flags;                trace(‘kmem_flags);
trace(*p);
```

Das linke Fragment erstellt eine globale D-Variable in Form des Zeigers `p`. Da das Objekt `kmem_flags` den Typ `int` aufweist, nimmt das Ergebnis von `&‘kmem_flags` den Typ `int *` an (d. h. ein Zeiger auf `int`). Das linke Fragment erfasst den Wert von `*p`, der den Zeiger bis zum Datenobjekt `kmem_flags` zurückverfolgt. Dieses Fragment ist deshalb gleichbedeutend mit dem rechten, das den Wert des Datenobjekts einfach über dessen Namen verfolgt.

Zeigersicherheit

Als C- oder C++-Programmierer hat Sie der vorangehende Abschnitt möglicherweise ein wenig erschreckt, da Sie wissen, dass die falsche Verwendung von Zeigern in einem Programm zum Programmabsturz führen kann. DTrace ist eine robuste, sichere Umgebung für die Ausführung von D-Programmen, in der diese Fehler keine Programmabstürze verursachen können. Es ist zwar möglich, dass Sie ein fehlerhaftes D-Programm schreiben, doch ungültige D-Zeigerzugriffe bewirken weder, dass DTrace noch der Betriebssystemkernel fehlschlagen

oder abstürzen. Stattdessen werden ungültige Zeigerzugriffe von der DTrace-Software erkannt, die Instrumentation deaktiviert und das Problem gemeldet, sodass Sie es beheben können.

Wenn Sie schon einmal ein Java-Programm geschrieben haben, wissen Sie wahrscheinlich, dass Java aus genau diesem Sicherheitsgrund keine Zeiger unterstützt. In D sind Zeiger jedoch erforderlich, da sie einen wesentlichen Bestandteil der Betriebssystemimplementierung in C darstellen. Deshalb bietet DTrace die gleiche Art von Sicherheitsmechanismen, die in der Programmiersprache Java die Beschädigung fehlerhafter Programme durch sich selbst oder andere Programme verhindern. Der Fehlermeldemechanismus in DTrace funktioniert ähnlich wie die Laufzeitumgebung in Java, die Programmierfehler erkennt und eine Ausnahme meldet.

Lassen Sie uns zur Betrachtung der Fehlerbehandlung und -meldung in DTrace absichtlich ein fehlerhaftes D-Programm mit Zeigern schreiben. Geben Sie das folgende D-Programm in einen Texteditor ein und speichern Sie es unter dem Namen `badptr.d`:

BEISPIEL 5–1 `badptr.d`: Veranschaulichung der Fehlerbehandlung in DTrace

```
BEGIN
{
    x = (int *)NULL;
    y = *x;
    trace(y);
}
```

Das Programm `badptr.d` erzeugt einen D-Zeiger namens `x`, der auf `int` zeigt. Das Programm weist dem Zeiger den speziellen, für ungültige Zeiger stehenden Wert `NULL` zu, der ein integrierter Aliasname für die Adresse 0 ist. Die Adresse 0 ist stets als ungültig vereinbart, sodass `NULL` in C- und D-Programmen dafür als Repräsentationswert eingesetzt werden kann. Mit einem Cast-Ausdruck wird `NULL` explizit in einen Zeiger auf eine Ganzzahl umgewandelt. Anschließend dereferenziert das Programm den Zeiger über den Ausdruck `*x` und weist das Ergebnis einer weiteren Variable `y` zu, bevor es schließlich die Ablaufverfolgung von `y` versucht. Wenn das D-Programm ausgeführt wird, erkennt DTrace bei der Ausführung der Anweisung `y = *x` einen ungültigen Zeigerzugriff und meldet den Fehler:

```
# dtrace -s badptr.d
dtrace: script '/dev/stdin' matched 1 probe
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address
(0x0) in action #2 at DIF offset 4
dtrace: 1 error on CPU 0
^C
#
```

Ein weiteres Problem, das durch Programme mit ungültigen Zeigern verursacht werden kann, ist ein *Ausrichtungsfehler*. Gemäß der Architekturkonvention werden essenzielle Datenobjekte wie Ganzzahlen im Speicher auf Grundlage ihrer Größe angeordnet. So werden beispielsweise

2-Byte-Ganzzahlen an Adressen ausgerichtet, die Vielfache von 2 sind, 4-Byte-Ganzzahlen an Vielfachen von 4 und so weiter. Wenn Sie einen Zeiger auf eine 4-Byte-Ganzzahl dereferenzieren und die Zeigeradresse ein ungültiger Wert und kein Vielfaches von 4 ist, schlägt der Zugriff mit einem Ausrichtungsfehler fehl. Ausrichtungsfehler in D weisen nahezu immer darauf hin, dass ein Zeiger aufgrund eines Fehlers im D-Programm einen ungültigen oder fehlerhaften Wert besitzt. Ändern Sie im Quellcode von `badptr.d` die Adresse `NULL` in `(int *)2` ab, um einen Ausrichtungsfehler zur Veranschaulichung zu erzeugen. Da `int` eine 4-Byte-Ganzzahl und 2 kein Vielfaches von 4 ist, verursacht der Ausdruck `*x` einen DTrace-Ausrichtungsfehler.

Näheres zum DTrace-Fehlermechanismus finden Sie unter „Der Prüfpunkt ERROR“ auf Seite 205.

Vektordeklarationen und Speicherung

D bietet neben den in Kapitel 3 beschriebenen dynamischen assoziativen Vektoren Unterstützung für *Skalare Vektoren*. Unter skalaren Vektoren versteht man eine Gruppe festgelegter Länge, die aufeinander folgender Speicheradressen enthält. In jeder dieser Adressen wird ein Wert vom gleichen Datentyp gespeichert. Auf skalare Vektoren wird durch Referenzierung auf die einzelnen Positionen mit einer Ganzzahl, ausgehend von Null, zugegriffen. Skalare Vektoren stimmen konzeptuell und syntaktisch direkt mit Vektoren in C und C++ überein. In D werden sie nicht so häufig eingesetzt wie assoziative Vektoren und ihre erweiterten Gegenstücke, die *Aggregate*, sind aber manchmal für den Zugriff auf vorhandene, in C deklarierte Betriebssystem-Vektordatenstrukturen nötig. Aggregate werden in Kapitel 9, „Aggregate“ beschrieben.

Ein skalarer Vektor von 5 Ganzzahlen ließe sich in D wie folgt mit dem Typ `int` deklarieren, wobei der Deklaration die Anzahl der Elemente in eckigen Klammern voranzustellen ist:

```
int a[5];
```

Das folgende Diagramm veranschaulicht die Vektorspeicherung:

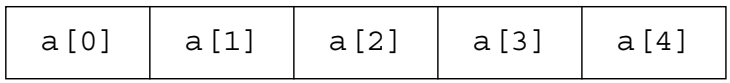

ABBILDUNG 5–1 Darstellung skalarer Vektoren

Der D-Ausdruck `a[0]` dient zur Referenzierung des ersten Vektorelements, `a[1]` referenziert das zweite und so weiter. Aus syntaktischer Sicht sind sich skalare und assoziative Vektoren sehr ähnlich. Sie können einen assoziativen Vektor von fünf Ganzzahlen, der mit einem Ganzzahlschlüssel referenziert wird, wie folgt deklarieren:

```
int a[int];
```

und diesen Vektor auch über den Ausdruck `a[0]` referenzieren. Aus Sicht der Speicherung und Implementation hingegen unterscheiden sich die beiden Vektoren sehr stark voneinander. Der statische Vektor `a` besteht aus fünf aufeinander folgenden, ab Null durchnummerierten Speicherpositionen und der Index verweist auf einen Versatz (Offset) in dem für den Vektor reservierten Speicherbereich. Dagegen haben assoziative Vektoren keine vordefinierte Größe und speichern Elemente nicht an aufeinander folgenden Speicherpositionen. Außerdem haben die Schlüssel assoziativer Vektoren keine Beziehung zur Speicherposition des entsprechenden Werts. Sie können auf die assoziativen Vektorelemente `a[0]` und `a[-5]` zugreifen, und DTrace reserviert nur zwei Speicherwörter, die aufeinander folgend sein können oder nicht. Bei den Schlüsseln assoziativer Vektoren handelt es sich um abstrakte Namen für die entsprechenden Werte, die keine Beziehung zu den Speicherpositionen der Werte haben.

Wenn Sie einen Vektor mit anfänglicher Zuweisung erstellen und einen einzelnen Ganzzahlausdruck als Vektorindex einsetzen (zum Beispiel `a[0] = 2`), erzeugt der D-Compiler stets einen neuen assoziativen Vektor, selbst wenn `a` in diesem Ausdruck auch als Zuweisung zu einem skalaren Vektor interpretiert werden könnte. In einer solchen Situation müssen skalare Vektoren vordeklariert werden, damit der D-Compiler die Definition der Vektorgröße erkennen und schließen kann, dass es sich um einen skalaren Vektor handelt.

Beziehung zwischen Zeigern und Vektoren

Ebenso wie in ANSI-C besteht auch in D eine besondere Beziehung zwischen Zeigern und Vektoren. Ein Vektor wird durch eine Variable dargestellt, der die Adresse ihrer ersten Speicherposition zugewiesen ist. Auch ein Zeiger ist die Adresse einer Speicherposition mit einem definierten Typ. In D ist also die Verwendung der Vektorindex-Notation [ ] sowohl für Zeiger- als auch Vektorvariablen zulässig. So haben beispielsweise die beiden folgenden D-Codefragmente dieselbe Bedeutung:

```
p = &a[0];                trace(a[2]);
trace(p[2]);
```

Im Fragment auf der linken Seite wird durch Anwendung des Operators `&` auf den Ausdruck `a[0]` der Zeiger `p` die Adresse des ersten Vektorelements in `a` zugewiesen. Der Ausdruck `p[2]` verfolgt den Wert des dritten Vektorelements (Index 2). Da `p` jetzt die auch `a` zugewiesene Adresse enthält, ergibt dieser Ausdruck denselben Wert wie `a[2]` im rechten Codefragment. Eine Konsequenz dieser Gleichbedeutung ist, dass C und D den Zugriff auf jeden Index eines beliebigen Zeigers oder Vektors zulassen. Weder der Compiler noch die DTrace-Laufzeitumgebung führen eine Überprüfung der Vektorgrenzen durch. Bei einem Zugriff auf eine Speicherposition außerhalb des vordefinierten Werts eines Vektors erhalten Sie entweder ein unerwartetes Ergebnis oder DTrace meldet wie im vorigen Beispiel einen Fehler aufgrund einer ungültigen Adresse. Wie immer, können Sie auch hierdurch weder DTrace noch das Betriebssystem beeinträchtigen, müssen aber den Fehler im D-Programm beseitigen.

Der Unterschied zwischen Zeigern und Vektoren besteht darin, dass eine Zeigervariable auf separaten Speicher verweist, der die ganzzahlige Adresse eines anderen Speichers enthält. Eine Vektorvariable benennt den Speicherbereich des Vektors selbst und nicht etwa die Position einer Ganzzahl, die ihrerseits die Position des Vektors enthält. Dieser Unterschied geht aus dem folgenden Schaubild hervor:

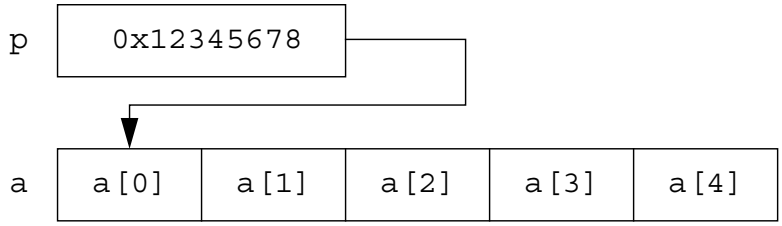

ABBILDUNG 5–2 Speicherung von Zeigern und Vektoren

Dieser Unterschied kommt in der D-Syntax zur Geltung, wenn Sie versuchen, Zeiger und skalare Vektoren zuzuweisen. Wenn `x` und `y` Zeigervariablen sind, ist der Ausdruck `x = y` zulässig. Er kopiert einfach die Zeigeradresse aus `y` an die mit `x` angegebene Speicherposition. Wenn `x` und `y` skalare Vektorvariablen sind, ist der Ausdruck `x = y` nicht zulässig. Eine Zuweisung von Vektoren als Ganze ist in D nicht möglich. Es können aber in jedem Kontext, in dem Zeiger zulässig sind, Vektorvariablen oder Symbolnamen verwendet werden. Handelt es sich bei `p` um einen Zeiger und bei `a` um einen Vektor, dann ist die Anweisung `p = a` erlaubt. Sie ist gleichbedeutend mit der Anweisung `p = &a[0]`.

Zeigerarithmetik

Da es sich bei Zeigern nur um Ganzzahlen handelt, die als Adressen für andere Objekte im Speicher verwendet werden, bietet D verschiedene Leistungsmerkmale für die Zeigerarithmetik. Diese unterscheidet sich jedoch von der Ganzzahlarithmetik. Bei der Zeigerarithmetik wird die betreffende Adresse implizit durch Multiplikation oder Division der Operanden mit der bzw. durch die Größe des Typs angepasst, auf den der Zeiger verweist. Das folgende D-Fragment demonstriert diese Eigenschaft:

```
int *x;

BEGIN
{
    trace(x);
    trace(x + 1);
    trace(x + 2);
}
```

Dieses Fragment erzeugt einen ganzzahligen Zeiger x und verfolgt dann seinen Wert, seinen um 1 erhöhten und seinen um 2 erhöhten Wert. Wenn Sie dieses Programm erstellen und ausführen, meldet DTrace die ganzzahligen Werte 0, 4 und 8.

Da x ein Zeiger auf ein Objekt des Typs int ist (Größe 4 Byte), wird der zugrunde liegende Zeigerwert durch Inkrementierung von x um 4 erhöht. Diese Eigenschaft erweist sich als nützlich, wenn Sie Zeiger zum Verweis auf aufeinander folgende Speicherpositionen wie zum Beispiel Vektoren einsetzen möchten. Wenn beispielsweise x der Adresse eines Vektors a, wie dem in Abbildung 5–2 dargestellten, zugewiesen würde, so wäre der Ausdruck `x + 1` gleichbedeutend mit dem Ausdruck `&a[1]`. Ebenso würde der Ausdruck `*(x + 1)` den Wert `a[1]` bezeichnen. Der D-Compiler implementiert Zeigerarithmetik immer dann, wenn ein Zeigerwert mit einem der Operatoren +=, + oder ++ inkrementiert wird.

Sie wird auch angewendet, wenn eine Ganzzahl von einem Zeiger auf der linken Seite subtrahiert, ein Zeiger von einem anderen Zeiger subtrahiert oder der Operator `--` auf einen Zeiger angewendet wird. So würde beispielsweise das folgende D-Programm das Ergebnis 2 verfolgen:

```
int *x, *y;
int a[5];

BEGIN
{
    x = &a[0];
    y = &a[2];
    trace(y - x);
}
```

Unspezifische Zeiger

Unter Umständen ist es praktisch, in einem D-Programm eine unspezifische (generische) Zeigeradresse darzustellen oder zu manipulieren, ohne den Datentyp anzugeben, auf den der Zeiger verweist. Unspezifische Zeiger können mit dem Typ `void *` angegeben werden, wobei das Schlüsselwort `void` für das Fehlen einer spezifischen Typangabe steht. Zu diesem Zweck kann ebenfalls der integrierte Typ-Aliasname `uintptr_t` verwendet werden, der einen vorzeichenlosen Integer-Typ einer für einen Zeiger im aktuellen Datenmodell geeigneten Größe darstellt. Auf Objekte des Typs `void *` darf Zeigerarithmetik nicht angewendet werden und diese Zeiger können nicht dereferenziert werden, ohne sie zuerst explizit in einen anderen Typ umzuwandeln. Wenn Sie Ganzzahlenarithmetik an einem Zeigerwert durchführen müssen, können Sie den Zeiger explizit in den Typ `uintptr_t` umwandeln.

Zeiger auf `void` können in jedem Kontext verwendet werden, in dem ein Zeiger auf einen anderen Datentyp erforderlich ist, wie beispielsweise bei einem Tupel-Ausdruck für einen assoziativen Vektor oder der rechten Seite einer Zuweisungsanweisung. Analog kann in einem Kontext, in dem ein Zeiger auf `void` benötigt wird, ein Zeiger eines beliebigen Datentyps

verwendet werden. Zum Einsetzen eines Zeigers auf einen nicht-`void`-Typ anstelle eines anderen nicht-`void`-Zeigertyps wird eine explizite Typumwandlung benötigt. Um Zeiger in Integer-Typen wie `uintptr_t` umzuwandeln oder diese Ganzzahlen in den entsprechenden Zeigertyp zurückzukonvertieren, müssen Sie stets eine explizite Typumwandlung vornehmen.

Mehrdimensionale Vektoren

Mehrdimensionale skalare Vektoren kommen in D nur selten zum Einsatz, stehen aber zum Zweck der Kompatibilität mit ANSI-C und zum Beobachten sowie Ansprechen von Betriebssystem-Datenstrukturen zur Verfügung, die anhand dieser Fähigkeit in C erstellt wurden. Ein mehrdimensionaler Vektor wird als eine zusammenhängende Folge skalarer Vektorgrößen in eckigen Klammern `[ ]` im Anschluss an den Basistyp deklariert. Um beispielsweise einen zweidimensionalen Vektor festgelegter Größe mit Ganzzahlen zu 12 Zeilen auf 34 Spalten zu deklarieren, würden Sie Folgendes schreiben:

```
int a[12][34];
```

Die gleiche Schreibweise gilt für den Zugriff auf mehrdimensionale skalare Vektoren. Um beispielsweise auf den in Zeile 0, Spalte 1 gespeicherten Wert zuzugreifen, würden Sie folgenden D-Ausdruck schreiben:

```
a[0][1]
```

Speicherpositionen für die Werte mehrdimensionaler skalarer Vektoren werden per Multiplikation der Zeilennummer durch die Gesamtzahl der deklarierten Spalten und anschließende Addition der Spaltennummer berechnet.

Achten Sie darauf, die Syntax für mehrdimensionale Vektoren nicht mit der D-Syntax für Zugriffe auf assoziative Vektoren zu verwechseln (d. h., `a[0][1]` ist nicht identisch mit `a[0, 1]`). Wenn Sie ein unvereinbares Tupel für einen assoziativen Vektor verwenden oder einen assoziativen Vektorzugriff auf einen skalaren Vektor versuchen, gibt der D-Compiler eine entsprechende Fehlermeldung aus und kompiliert das Programm nicht.

Zeiger auf DTrace-Objekte

Der D-Compiler verbietet Ihnen den Einsatz des Operators & zum Erhalten von Zeigern auf DTrace-Objekte wie assoziative Vektoren, integrierte Funktionen und Variablen. Sie werden daran gehindert, die Adressen dieser Variablen abzurufen, damit die DTrace-Laufzeitumgebung sie zwischen den verschiedenen Prüfpunktauslösungen nach Bedarf verschieben und so den für Ihre Programme benötigten Speicher effizienter verwalten kann. Wenn Sie zusammengesetzte Strukturen erstellen, besteht die Möglichkeit, Ausdrücke zu formulieren, die die Kerneladresse des DTrace-Objektspeichers abrufen. Verzichten Sie in

Ihren D-Programmen auf solche Ausdrücke. Sollten Sie dennoch einen solchen Ausdruck benötigen, achten Sie darauf, die Adresse nicht über mehrere Prüfpunktauslösungen hinweg zwischenzuspeichern.

In ANSI-C können Zeiger auch für indirekte Funktionsaufrufe oder Zuweisungen wie das Absetzen eines Ausdrucks mit dem unären *-Dereferenzierungsoperator auf der linken Seite eines Zuweisungsoperators verwendet werden. In D sind diese Arten der Ausdrücke mit Zeigern nicht erlaubt. Sie dürfen D-Variablen Werte nur direkt zuweisen, indem Sie entweder ihre Namen verwenden oder den Vektorindex-Operator `[]` auf einen skalaren oder assoziativen D-Vektor anwenden. In der DTrace-Umgebung definierte Funktionen dürfen, wie in Kapitel 10, „Aktionen und Subroutinen“ erläutert, nur über ihren Namen aufgerufen werden. Indirekte Funktionsaufrufe mithilfe von Zeigern sind in D nicht zulässig.

Zeiger und Adressräume

Ein Zeiger ist eine Adresse, die innerhalb eines *virtuellen Adressraums* eine Übersetzung für einen physischen Speicherbereich liefert. DTrace führt D-Programme innerhalb des Adressraums für den Betriebssystemkernel selbst aus. Das Solaris-System verwaltet zahlreiche Adressräume: davon einen für den Betriebssystemkernel und einen für Benutzerprozesse. Da jeder Adressraum scheinbar auf den gesamten Speicher des Systems zugreifen kann, lässt sich derselbe virtuelle Adresszeigerwert in den verschiedenen Adressräumen wieder verwenden und in unterschiedliche physische Speicherbereiche übersetzen. Deshalb müssen Sie beim Schreiben von D-Programmen die Adressräume der Zeiger beachten, die Sie verwenden möchten.

Wenn Sie beispielsweise mit dem Provider `syscall` den Eintritt in einen Systemaufruf instrumentieren, der einen Zeiger auf eine Ganzzahl oder einen Vektor von Ganzzahlen als Argument annimmt (z. B. `pipe(2)`), ist eine Dereferenzierung des Zeigers oder Vektors mit einem der Operatoren * oder `[]` nicht zulässig, da sich die betreffende Adresse in dem Adressraum des Benutzerprozesses befindet, der den Systemaufruf durchgeführt hat. In D würde die Anwendung des Operators * oder `[]` auf diese Adresse einen Zugriff auf den Kernel-Adressraum bedeuten, der einen Fehler aufgrund ungültiger Adresse oder die Rückgabe unerwarteter Daten an das D-Programm zur Folge hätte. Dies hängt davon ab, ob die Adresse zufällig mit einer gültigen Kernel-Adresse übereinstimmt oder nicht.

Um über einen DTrace-Prüfpunkt auf den Benutzerprozess-Speicher zuzugreifen, müssen Sie eine der in Kapitel 10, „Aktionen und Subroutinen“ beschriebenen Funktionen `copyin()`, `copyinstr()` oder `copyinto()` auf den Benutzer-Adressraumzeiger anwenden. Achten Sie beim Schreiben von D-Programmen darauf, Variablen für Benutzeradressen angemessen zu benennen und zu kommentieren. Sie können dadurch einiges an Durcheinander verhindern. Alternativ können Sie Benutzeradressen als `uintptr_t` speichern und somit der Gefahr aus dem Weg gehen, dass Sie D-Code schreiben, der diese dereferenziert. Verfahren zum Verwenden von DTrace für Benutzerprozesse sind in Kapitel 33, „Ablaufverfolgung von Benutzerprozessen“ beschrieben.

◆ ◆ ◆ KAPITEL 6

Zeichenketten

DTrace bietet Unterstützung für die Ablaufverfolgung und Manipulation von Zeichenketten. In diesem Kapitel werden sämtliche Leistungsmerkmale der Programmiersprache D zum Deklarieren und Manipulieren von Zeichenketten besprochen. Im Gegensatz zu ANSI-C besitzen Zeichenketten in D eine eigene, integrierte Typ- und Operatorunterstützung, sodass sie sich einfach und eindeutig in Tracing-Programme einbauen lassen.

Zeichenkettendarstellung

Zeichenketten werden in DTrace als Vektoren von Zeichen mit einem abschließenden Null-Byte (d. h. einem Byte mit dem Wert Null, das in der Regel in der Form `'\0'` geschrieben wird) dargestellt. Der sichtbare Teil einer Zeichenkette hat eine variable Länge, die von der Position des Null-Bytes abhängt. DTrace speichert jedoch jede Zeichenkette in einem Vektor festgelegter Größe, sodass alle Prüfpunkte eine einheitliche Datenmenge verfolgen. Zwar dürfen die Zeichenketten diese vordefinierte Höchstlänge nicht überschreiten, doch lässt sich die Höchstlänge im D-Programm oder in der `dtrace`-Befehlszeile durch Anpassung der Option `strsize` ändern. Weitere Informationen zu abstimmbaren DTrace-Optionen finden Sie in Kapitel 16, „Optionen und Tunables “. Die Standardhöchstlänge für Zeichenketten beträgt 256 Byte.

Die Sprache D bietet einen expliziten `string`-Typ, der anstelle des Typs `char *` zum Verweisen auf Zeichenketten genutzt werden kann. Der Typ `string` ist insoweit gleichbedeutend mit einem `char *`, als er die Adresse einer Folge von Zeichen darstellt. Bei Anwendung auf Ausdrücke des Typs `string()` stellen der D-Compiler und D-Funktionen wie `trace` jedoch erweiterte Fähigkeiten bereit. So beseitigt der String-Typ beispielsweise die Zweideutigkeit des Typs `char *` für den Fall, dass Sie die tatsächlichen Byte einer Zeichenkette verfolgen müssen. In der D-Anweisung:

```
trace(s);
```

verfolgt DTrace, wenn `s` den Typ `char *` aufweist, den Wert des Zeigers `s` (d. h., es wird ein ganzzahliger Adresswert aufgezeichnet). In der D-Anweisung:

```
trace(*s);
```

dereferenziert der D-Compiler gemäß der Definition des Operators * den Zeiger `s` und verfolgt das einzelne Zeichen an jener Position. Dieses Verhalten ist grundlegend für die Manipulation von Zeichenzeigern, die entwurfsgemäß entweder auf einzelne Zeichen oder auf Vektoren mit byte-großen Ganzzahlen verweisen, die keine Zeichenketten sind und nicht mit einem Null-Byte enden. In der D-Anweisung:

```
trace(s);
```

teilt der Typ `string`, wenn `s` ein `string` ist, dem D-Compiler mit, dass DTrace eine Zeichenkette verfolgen soll, deren Adresse in der Variable `s` gespeichert ist. Außerdem können Sie lexikalische Vergleiche von Ausdrücken des Typs `string` durchführen. Dies wird unter „Zeichenkettenvergleich“ auf Seite 95 beschrieben.

Konstante Zeichenketten

Konstante Zeichenketten sind von doppelten Anführungszeichen (") umgeben und erhalten vom D-Compiler automatisch den Typ `string`. Sie können konstante Zeichenketten beliebiger Länge definieren, die nur durch die Speicherkapazität begrenzt ist, die DTrace auf dem System nutzen darf. Das abschließende Null-Byte (`\0`) wird vom D-Compiler automatisch an jede von Ihnen deklarierte konstante Zeichenkette angefügt. Die Größe eines konstanten Zeichenkettenobjekts ergibt sich aus der Byteanzahl der Zeichenkette plus einem zusätzlichen Byte für das abschließende Null-Byte.

Zeichenkettenkonstanten dürfen keine wörtlichen Zeilentrenner enthalten. Zur Angabe einer neuen Zeile innerhalb einer Zeichenkette verwenden Sie stattdessen die Ersatzdarstellung `\n`. Alle bereits in Tabelle 2–5 für Zeichenkonstanten definierten Sonderzeichen-Ersatzdarstellungen dürfen auch in konstanten Zeichenketten vorkommen.

Zeichenkettenzuweisung

Anders als die Zuweisung von `char *`-Variablen werden Zeichenketten nicht per Verweis, sondern per Wert zugewiesen. Die Zeichenkettenzuweisung erfolgt mit dem Operator =. Dabei werden die tatsächlichen Byte der Zeichenkette ab dem Quelloperanden bis zu einschließlich dem Null-Byte in die Variable auf der linken Seite kopiert, bei der es sich um den Typ `string` handeln muss. Sie können eine neue Variable des Typs `string` erzeugen, indem Sie ihr einen Ausdruck des Typs `string` zuweisen. So würde beispielsweise die D-Anweisung:

```
s = "hello";
```

eine neue Variable `s` des Typs `string` erzeugen und die 6 Byte der Zeichenkette `"hello"` in sie kopieren (5 druckbare Zeichen plus Null-Byte). Die Zeichenkettenzuweisung verhält sich

analog zur C-Bibliotheksfunktion strcpy(3C) mit der Abweichung, dass sich bei Überschreiten der Speicherkapazität der Zielzeichenkette durch die Ausgangszeichenkette eine Zeichenkette ergibt, die automatisch an diesem Grenzwert abgeschnitten wird.

Einer Zeichenkettenvariable kann auch ein Ausdruck eines mit Zeichenketten verträglichen Typs zugewiesen werden. In diesem Fall erweitert der D-Compiler den Ausgangsausdruck automatisch auf den String-Typ und führt eine Zeichenkettenzuweisung durch. Der D-Compiler lässt für beliebige Ausdrücke des Typs `char *` oder `char[`*n*`]` (d. h. einen skalaren Vektor des Typs `char` mit beliebiger Größe) eine Erweiterung auf `string` zu.

Zeichenkettenumwandlung

Ausdrücke anderer Typen können explizit in den Typ `string` umgewandelt werden. Hierzu verwenden Sie einen Cast-Ausdruck oder wenden den speziellen Operator `stringof` an. Die beiden Verfahren sind äquivalent:

`s = (string)` *Ausdruck* `s = stringof (` *Ausdruck* `)`

Der Operator `stringof` hat eine sehr starke Bindung an den Operanden rechts neben ihm. In der Regel wird der Ausdruck zur Verdeutlichung in Klammern gesetzt, die allerdings nicht wirklich erforderlich sind.

Jeder Ausdruck skalaren Typs wie zum Beispiel Zeiger oder Ganzzahlen oder skalare Vektoradressen können in string umgewandelt werden. Ausdrücke anderer Typen wie `void` lassen sich nicht in `string` konvertieren. Wenn Sie eine ungültige Adresse fälschlicherweise in einen String-Typ umwandeln, verhindern die DTrace-Sicherheitseinrichtungen zwar eine Beschädigung des Systems oder von DTrace, doch wird unter Umständen trotzdem eine Sequenz nicht entzifferbarer Zeichen verfolgt.

Zeichenkettenvergleich

D überlädt die binären relationalen Operatoren und erlaubt ihren Einsatz für Zeichenketten- sowie Ganzzahlenvergleiche. Die relationalen Operatoren führen einen Zeichenkettenvergleich durch, wenn beide Operanden den Typ `string` besitzen oder wenn ein Operand ein `string` ist und der andere Operand, wie unter `beschrieben, auf den Typ` string (siehe „Zeichenkettenzuweisung“ auf Seite 94) erweitert werden kann. Zum Vergleichen von Zeichenketten können alle relationalen Operatoren verwendet werden:

TABELLE 6–1 Relationale Operatoren für Zeichenketten in D

<	Operand auf linker Seite ist kleiner als Operand auf rechter Seite

TABELLE 6–1 Relationale Operatoren für Zeichenketten in D *(Fortsetzung)*

`<=`	Operand auf linker Seite ist kleiner als oder gleich dem Operanden auf rechter Seite
`>`	Operand auf linker Seite ist größer als Operand auf rechter Seite
`>=`	Operand auf linker Seite ist größer als oder gleich dem Operanden auf rechter Seite
`==`	Operand auf linker Seite ist gleich dem Operanden auf rechter Seite
`!=`	Operand auf linker Seite ist ungleich dem Operanden auf rechter Seite

Wie auch bei Ganzzahlen ergibt jeder Operator einen Wert des Typs `int`, der gleich 1 ist, wenn die Bedingung wahr ist und 0, wenn sie falsch ist.

Die relationalen Operatoren vergleichen die beiden eingegebenen Zeichenketten Byte für Byte auf ähnliche Weise wie die C-Bibliotheksroutine `strcmp(3C)`. Jedes Byte wird so lange über seinen entsprechenden ganzzahligen Wert im ASCII-Zeichensatz (siehe `ascii(5)`) verglichen, bis ein Null-Byte gelesen wird oder die maximale Zeichenkettenlänge erreicht ist. Sehen Sie hier einige Beispiele für Zeichenkettenvergleiche in D und ihre Ergebnisse:

`"coffee" < "espresso"`	... gibt 1 (wahr) zurück
`"coffee" == "coffee"`	... gibt 1 (wahr) zurück
`"coffee" >= "mocha"`	... gibt 0 (falsch) zurück

◆ ◆ ◆ KAPITEL 7

Strukturen und Unionen

Ansammlungen zusammenhängender Variablen können in Datenobjekten gruppiert werden, die als *Strukturen* (structs) und *Unionen* (unions) bezeichnet werden. In D lassen sich diese Objekte definieren, indem Sie neue Typen für sie vereinbaren. Sie können Ihre neuen Typen für beliebige D-Variablen, einschließlich der Werte assoziativer Vektoren, verwenden. In diesem Kapitel betrachten wir die Syntax und Semantik für die Erzeugung und Manipulation dieser zusammengesetzten Typen sowie die D-Operatoren, die mit ihnen interagieren. Die Syntax für Strukturen und Unionen wird anhand verschiedener Beispielprogramme dargestellt, die den Nutzen der DTrace-Provider `fbt` und `pid` demonstrieren.

Strukturen

Das D-Schlüsselwort `struct`, Abkürzung für *structure* (Struktur), dient zur Einführung eines neuen, aus einer Gruppe anderer Typen zusammengesetzten Typs. Der neue Struct-Typ kann als Typ für D-Variablen und Vektoren verwendet werden, sodass Sie Gruppen zusammenhängender Variablen unter einem einzigen Namen gruppieren können. D-Strukturen sind mit den entsprechenden Konstrukten in C und C++ identisch. An Java gewöhnte Programmierer sollten sich eine D-Struktur wie eine Klasse (class) mit Datenmitgliedern, aber ohne Methoden vorstellen.

Nehmen wir an, es soll ein anspruchsvolleres D-Programm für die Ablaufverfolgung von Systemaufrufen geschrieben werden, das verschiedene Informationen über jeden von der Shell ausgeführten `read(2)` und `write(2)` Systemaufruf aufzeichnet. Bei diesen Informationen kann es sich um die verstrichene Zeit, die Anzahl der Aufrufe oder die größte als Argument übergebene Byteanzahl handeln. Sie könnten, wie in folgendem Beispiel, eine D-Klausel zum Aufzeichnen dieser Eigenschaften in drei separaten assoziativen Vektoren schreiben:

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{
    ts[probefunc] = timestamp;
```

```
    calls[probefunc]++;
    maxbytes[probefunc] = arg2 > maxbytes[probefunc] ?
        arg2 : maxbytes[probefunc];
}
```

Diese Klausel ist jedoch nicht effizient, da DTrace drei separate assoziative Vektoren erzeugen und für jeden eine separate Kopie der identischen Tupelwerte von `probefunc` speichern muss. Wenn Sie stattdessen eine Struktur verwenden, können Sie nicht nur Platz sparen, sondern erhalten auch ein Programm, das sich leichter lesen und pflegen lässt. Deklarieren Sie zu Beginn der Programmquelldatei zuerst einen neuen Struct-Typ:

```
struct callinfo {
    uint64_t ts;      /* timestamp of last syscall entry */
    uint64_t elapsed; /* total elapsed time in nanoseconds */
    uint64_t calls;   /* number of calls made */
    size_t maxbytes;  /* maximum byte count argument */
};
```

Auf das Schlüsselwort `struct` folgt ein optionaler Bezeichner zum Verweis auf unseren neuen Typ, der nun als `struct callinfo` bekannt ist. Anschließend sind die Komponenten der Struktur in geschweiften Klammern `{ }` aufgeführt und die gesamte Deklaration endet mit einem Strichpunkt (`;`). Die Definition der einzelnen Strukturkomponenten (members) erfolgt in der Syntax für D-Variablendeklarationen. Dabei steht zuerst der Typ der Komponente, gefolgt von einem Bezeichner, der die Komponente benennt, und einem weiteren Strichpunkt (`;`).

Die Strukturdeklaration selbst definiert einfach den neuen Typ; sie erzeugt weder eine Variable noch reserviert sie Speicherplatz in DTrace. Nach einmaliger Deklaration können Sie `struct callinfo` im gesamten Rest des D-Programms als Typ verwenden, und jede Variable des Typs `struct callinfo` speichert eine Kopie der vier mit unserer Strukturvorlage beschriebenen Variablen. Die Komponenten werden im Speicher gemäß der Komponentenliste und mit dem ggf. für die Datenobjektausrichtung benötigten Zwischenraum zwischen den einzelnen Komponenten angeordnet.

Für den Zugriff auf die einzelnen Komponentenwerte mit dem Operator „`.`" können Sie die Komponentennamen verwenden. Schreiben Sie hierzu einen Ausdruck in der Form:

Variablenname`.`*Komponentenname*

Das folgende Beispiel zeigt ein verbessertes Programm mit dem neuen Strukturtyp. Geben Sie dieses D-Programm in einen Texteditor ein und speichern Sie es unter dem Namen `rwinfo.d`:

BEISPIEL 7–1 `rwinfo.d`: : Erfassen statistischer Daten über `read`(2) und `write`(2)

```
struct callinfo {
    uint64_t ts;      /* timestamp of last syscall entry */
    uint64_t elapsed; /* total elapsed time in nanoseconds */
```

BEISPIEL 7–1 rwinfo.d: : Erfassen statistischer Daten über read(2) und write(2) *(Fortsetzung)*

```
    uint64_t calls;   /* number of calls made */
    size_t maxbytes;  /* maximum byte count argument */
};

struct callinfo i[string];    /* declare i as an associative array */

syscall::read:entry, syscall::write:entry
/pid == $1/
{
    i[probefunc].ts = timestamp;
    i[probefunc].calls++;
    i[probefunc].maxbytes = arg2 > i[probefunc].maxbytes ?
        arg2 : i[probefunc].maxbytes;
}

syscall::read:return, syscall::write:return
/i[probefunc].ts != 0 && pid == $1/
{
    i[probefunc].elapsed += timestamp - i[probefunc].ts;
}

END
{
    printf("        calls  max bytes  elapsed nsecs\n");
    printf("------  -----  ---------  -------------\n");
    printf("  read  %5d  %9d  %d\n",
        i["read"].calls, i["read"].maxbytes, i["read"].elapsed);
    printf(" write  %5d  %9d  %d\n",
        i["write"].calls, i["write"].maxbytes, i["write"].elapsed);
}
```

Wenn Sie das Programm eingegeben haben, führen Sie `dtrace -q -s rwinfo.d` aus und geben dabei einen Ihrer Shell-Prozesse an. Geben Sie dann den ein oder anderen Befehl in die Shell ein und drücken Sie abschließend im `dtrace`-Terminal die Tastenkombination Strg-C, um den Prüfpunkt `END` auszulösen und die Ergebnisse anzeigen zu lassen:

```
# dtrace -q -s rwinfo.d ‘pgrep -n ksh‘
^C
        calls  max bytes  elapsed nsecs
------  -----  ---------  -------------
  read     36       1024  3588283144
 write     35         59  14945541
#
```

Zeiger auf Strukturen

Verweise auf Strukturen über Zeiger sind in C und D sehr üblich. Für den Zugriff auf Strukturkomponenten über einen Zeiger steht der Operator `->` zur Verfügung. Besitzt `struct s` eine Komponente `m` und es liegt ein Zeiger namens `sp` auf diese Struktur vor (d. h. `sp` ist eine Variable des Typs `struct s *`), können Sie entweder den `*`-Operator einsetzen, um den Zeiger `sp` für den Zugriff auf die Komponente zunächst zu dereferenzieren:

```
struct s *sp;

(*sp).m
```

oder Sie können diese Schreibweise anhand des Operators `->` kürzer fassen. Die beiden folgenden D-Fragmente sind gleichbedeutend, sofern `sp` ein Zeiger auf eine Struktur ist:

```
(*sp).m                 sp->m
```

DTrace umfasst mehrere integrierte Variablen, die Zeiger auf Strukturen darstellen. Beispiele sind `curpsinfo` und `curlwpsinfo`. Diese Zeiger verweisen auf die Strukturen `psinfo` bzw. `lwpsinfo` und ihr Inhalt bietet eine Momentaufnahme der Informationen über den Zustand des aktuellen Prozesses und leichtgewichtigen Prozesses (LWP) des Threads, der den aktuellen Prüfpunkt ausgelöst hat. Ein Solaris-LWP ist die Kerneldarstellung eines Benutzer-Threads, auf dem die Solaris- und POSIX-Thread-Schnittstellen aufbauen. Der Einfachheit halber exportiert DTrace diese Informationen in derselben Form wie die `/proc`-Dateisystemdateien `/proc/`*`PID`*`/psinfo` und `/proc/`*`PID`*`/lwps/`*`LWPID`*`/lwpsinfo`. Die·`/proc` -Strukturen werden von Überwachungs- und Debugging-Tools wie `ps(1)`, `pgrep(1)` und `truss(1)` genutzt, sind in der Systemdefinitionsdatei `<sys/procfs.h>` definiert und in der Manpage `proc(4)` beschrieben. Beispiele für Ausdrücke mit `curpsinfo`, ihre Typen und Bedeutung sind:

`curpsinfo->pr_pid`	`pid_t`	aktuelle Prozess-ID
`curpsinfo->pr_fname`	`char []`	Name der ausführbaren Datei
`curpsinfo->pr_psargs`	`char []`	anfängliche Befehlszeilenargumente

Sie sollten sich die vollständige Strukturdefinition später noch einmal genauer ansehen. Untersuchen Sie dazu die Definitionsdatei `<sys/procfs.h>` und die entsprechenden Beschreibungen in `proc(4)`. Im nächsten Beispiel wird mit der Komponente `pr_psargs` durch einen Vergleich von Befehlszeilenargumenten ein bestimmter Prozess angegeben.

Strukturen werden häufig zum Erzeugen komplexer Datenstrukturen in C-Programmen eingesetzt. Folglich stellt die Fähigkeit, Strukturen aus D zu beschreiben und zu referenzieren, eine leistungsfähige Einrichtung zur Beobachtung der inneren Abläufe des Solaris-Betriebssystemkernels und seiner Systemschnittstellen dar. Der nächste Beispielcode enthält nicht nur die bereits erwähnte Struktur `curpsinfo`, sondern untersucht auch einige

Kernelstrukturen, indem er die Beziehung zwischen dem Treiber ksyms(7D) und den read(2)-Anforderungen beobachtet. Zum Reagieren auf die Anforderungen von Lesezugriffen auf die zeichenorientierte Gerätedatei /dev/ksyms stützt sich der Treiber auf die häufig verwendeten Strukturen uio(9S) und iovec(9S).

Die uio-Struktur, auf die über den Namen struct uio oder den Typ-Aliasnamen uio_t zugegriffen wird, ist in der Manpage uio(9S) beschrieben. Sie dient zum Beschreiben von E/A-Anforderungen, für die Daten zwischen dem Kernel und einem Benutzerprozess kopiert werden müssen. uio enthält selbst einen Vektor einer oder mehrerer iovec(9S)-Strukturen, die jeweils einen Teil der E/A-Anforderung beschreiben, falls unter Verwendung der Systemaufrufe readv(2) oder writev(2) mehrere Chunks erforderlich sein sollten. Eine der Kernel-DDI-Routinen (DDI = Gerätetreiberschnittstelle), die auf struct uio einwirken, ist die Funktion uiomove(9F). Sie gehört zu einer Familie von Funktionen, derer sich Kerneltreiber bedienen, um auf read(2)-Anforderungen für Benutzerprozesse zu antworten und Daten in Benutzerprozesse zurückzukopieren.

Der Treiber ksyms verwaltet eine zeichenorientierte Gerätedatei namens /dev/ksyms. Dabei handelt es sich scheinbar um eine ELF-Datei mit Informationen über die Symboltabelle des Kernels, die aber tatsächlich nur eine vom Treiber anhand der derzeit im Kernel geladenen Module erzeugte „Illusion“ ist. uiomove(9F)-Anforderungen reagiert der Treiber mit der read(2)-Routine. Das nächste Beispiel veranschaulicht, wie die Argumente und Aufrufe von read(2) durch /dev/ksyms die Treiberaufrufe von uiomove(9F) abgleichen, um das Ergebnis wieder in den Benutzeradressraum an der für read(2) angegebenen Position zu kopieren.

Mit dem Dienstprogramm strings(1); und der Option -a können zahlreiche Lesezugriffe durch /dev/ksyms erzwungen werden . Führen Sie strings -a /dev/ksyms in Ihrer Shell aus und betrachten Sie die Ausgabe. Geben Sie in einen Texteditor die erste Klausel des Beispielskripts ein und speichern Sie es unter dem Namen ksyms.d:

```
syscall::read:entry
/curpsinfo->pr_psargs == "strings -a /dev/ksyms"/
{
    printf("read %u bytes to user address %x\n", arg2, arg1);
}
```

In dieser ersten Klausel kommt der Ausdruck curpsinfo->pr_psargs zum Zugriff auf und den Vergleich der Befehlszeilenargumente unseres Befehls strings(1) vor. So wird gewährleistet, dass das Skript vor der Verfolgung der Argumente die richtigen read(2)-Anforderungen auswählt. Beachten Sie: Durch Verwendung des Operators == mit einem Argument auf der linken Seite, das ein Vektor des Typs char ist, und einem Argument auf der rechten Seite, bei dem es sich um eine Zeichenkette handelt, schließt der D-Compiler, dass das linke Argument auf den String-Typ erweitert und ein Zeichenkettenvergleich durchgeführt werden soll. Geben Sie den Befehl dtrace -q -s ksyms.d ein und führen Sie ihn in einer Shell aus. Geben Sie dann in eine andere Shell den Befehl strings -a /dev/ksyms ein. Während der Ausführung von strings(1) generiert DTrace eine Ausgabe wie diese:

```
# dtrace -q -s ksyms.d
read 8192 bytes to user address 80639fc
read 8192 bytes to user address 80639fc
read 8192 bytes to user address 80639fc
read 8192 bytes to user address 80639fc
...
^C
#
```

Dieses Beispiel lässt sich mithilfe einer gebräuchlichen D-Programmiertechnik so ausweiten, dass ein Thread von dieser anfänglichen `read(2)`-Anforderung bis tief in den Kernel verfolgt werden kann. Bei Eintritt des Kernels in `syscall::read:entry` setzt das nächste Skript eine thread-lokale Flag-Variable, die angibt, dass dieser Thread von Interesse ist und bei `syscall::read:return` wieder gelöscht wird. Das einmal gesetzte Flag kann als Prädikat für andere Prüfpunkte zur Instrumentierung von Kernelfunktionen wie zum Beispiel `uiomove(9F)` verwendet werden. Der DTrace-Provider `fbt` (function boundary tracing) veröffentlicht Eintritt- und Rückkehr-Prüfpunkte für Funktionen, die im Kernel definiert sind, einschließlich jener in der Gerätetreiberschnittstelle. Geben Sie folgenden Quellcode ein, in dem der Provider `fbt`zur Instrumentierung von `uiomove(9F)` eingesetzt wird, und speichern Sie ihn unter dem Namen `ksyms.d`:

BEISPIEL 7–2 `ksyms.d`: Beziehung zwischen Ablaufverfolgung von `read(2)` und `uiomove(9F)`

```
/*
 * When our strings(1) invocation starts a read(2), set a watched flag on
 * the current thread.  When the read(2) finishes, clear the watched flag.
 */
syscall::read:entry
/curpsinfo->pr_psargs == "strings -a /dev/ksyms"/
{
    printf("read %u bytes to user address %x\n", arg2, arg1);
    self->watched = 1;
}

syscall::read:return
/self->watched/
{
    self->watched = 0;
}

/*
 * Instrument uiomove(9F).  The prototype for this function is as follows:
 * int uiomove(caddr_t addr, size_t nbytes, enum uio_rw rwflag, uio_t *uio);
 */
fbt::uiomove:entry
/self->watched/
{
```

BEISPIEL 7–2 `ksyms.d`: Beziehung zwischen Ablaufverfolgung von `read`(2) und `uiomove`(9F) *(Fortsetzung)*

```
    this->iov = args[3]->uio_iov;

    printf("uiomove %u bytes to %p in pid %d\n",
        this->iov->iov_len, this->iov->iov_base, pid);
}
```

Die abschließende Klausel dieses Beispiels verwendet die thread-lokale Variable `self->watched`, um festzustellen, wann ein bestimmter Kernel-Thread die DDI-Routine `uiomove`(9F) betritt. An diesem Punkt angelangt, greift das Skript mithilfe des integrierten `args`-Vektors auf das vierte Argument (`args[3]`) von `uiomove()` zu, das ein Zeiger auf `struct uio` ist, die wiederum die Anforderung darstellt. Der D-Compiler weist jedem Element des `args`-Vektors automatisch den dem C-Funktionsprototypen für die instrumentierte Kernel-Routine entsprechenden Typ zu. Die `uio_iov`-Komponente enthält einen Zeiger auf die Struktur `struct iovec` für die Anforderung. Zur Verwendung in unserer Klausel wird eine Kopie dieses Zeigers in der klausel-lokalen Variable `this->iov` gespeichert. In der letzten Anweisung dereferenziert das Skript `this->iov`, um auf die `iovec`-Komponenten `iov_len` und `iov_base` zuzugreifen, die die Bytelänge bzw. die Ziel-Basisadresse von `uiomove`(9F) darstellen. Diese Werte sollten mit den Eingabeparametern des auf dem Treiber ausgeführten Systemaufrufs `read`(2) übereinstimmen. Wechseln Sie zu Ihrer Shell und führen Sie `dtrace -q -s ksyms.d` aus. Geben Sie anschließend wieder den Befehl `strings -a /dev/ksyms` in eine andere Shell ein. Es müsste eine Ausgabe wie im folgenden Beispiel produziert werden:

```
# dtrace -q -s ksyms.d
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
read 8192 bytes at user address 80639fc
uiomove 8192 bytes to 80639fc in pid 101038
...
^C
#
```

Die Adressen und Prozess-IDs in Ihrer Ausgabe weichen zwar hiervon ab, doch sollten Sie beobachten, dass die für `read`(2) eingegebenen Argumente mit den vom Treiber `ksyms` an `uiomove`(9F) übergebenen Parametern übereinstimmen.

Unionen

Unionen sind eine weitere Art der von ANSI-C und D unterstützten zusammengesetzten Typen. Sie sind eng mit Strukturen verwandt. Eine Union ist ein zusammengesetzter Typ, in dem Alternativen (Komponenten) unterschiedlicher Typen definiert sind und alle Alternativenobjekte denselben Speicherbereich belegen. Sie ist folglich ein Objekt eines veränderlichen Typs, in dem, abhängig von der Zuweisung der Union, zu einem gegebenen Zeitpunkt nur je eine Alternative gültig ist. In der Regel wird die aktuelle gültige Unionsalternative durch eine andere Variable oder ein Statuselement angegeben. Eine Union hat die Größe ihrer größten Alternative, und die für sie verwendete Speicherausrichtung entspricht der maximalen für die Unionsalternativen benötigten Ausrichtung.

Das Solaris-Framework `kstat` definiert eine Struktur, die die in folgendem Beispiel zur Veranschaulichung von C- und D-Unionen verwendete Union enthält. Das `kstat`-Framework dient zum Exportieren eines Satzes benannter Zähler, die Kernel-Statistiken wie die Speichernutzung und den E/A-Datendurchsatz darstellen. Mit dem Framework werden Dienstprogramme wie `mpstat(1M)` und `iostat(1M)` implementiert. In diesem Framework stellt `struct kstat_named` einen benannten Zähler und dessen Werte dar. Es ist wie folgt definiert:

```
struct kstat_named {
    char name[KSTAT_STRLEN]; /* name of counter */
    uchar_t data_type;    /* data type */
    union {
        char c[16];
        int32_t i32;
        uint32_t ui32;
        long l;
        ulong_t ul;
        ...
    } value;    /* value of counter */
};
```

Die untersuchte Deklaration ist aus Gründen der Einfachheit abgekürzt. Die vollständige Strukturdefinition befindet sich in der Definitionsdatei `<sys/kstat.h>` und ist in `kstat_named(9S)` beschrieben. Die obige Deklaration ist sowohl in ANSI-C als auch in D gültig. Sie definiert eine Struktur, zu deren Komponenten ein Unionswert mit Alternativen unterschiedlicher Typen zählt, die vom Typ des Zählers abhängen. Beachten Sie, dass auf einen formalen Namen des Unionstyps verzichtet wurde, da die Union selbst innerhalb eines anderen Typs, `struct kstat_named`, deklariert ist. Dieser Deklarationsstil wird als *anonyme Union* bezeichnet. Die Komponente namens `value` nimmt den in der vorangehenden Deklaration beschriebenen Union-Typ an, aber dieser Union-Typ selbst hat keinen Namen, da er an keiner anderen Stelle benutzt werden muss. Der Strukturkomponente `data_type` wird ein Wert zugewiesen, der angibt, welche Alternative der Union für die einzelnen Objekte des Typs `struct kstat_named` gültig ist. Als Werte für `data_type` sind verschiedene

C-Preprozessor-Symbole definiert. So ist beispielsweise das Symbol `KSTAT_DATA_CHAR` gleich Null und gibt an, dass sich die Alternative `value.c` an der derzeitigen Speicherposition des Werts befindet.

Beispiel 7–3 zeigt den Zugriff auf die Union `kstat_named.value` durch die Ablaufverfolgung eines Benutzerprozesses. Die `kstat`-Zähler können von Benutzerprozessen mithilfe der Funktion `kstat_data_lookup(3KSTAT)` geprüft werden, die einen Zeiger auf `struct kstat_named` zurückgibt. Bei seiner Ausführung ruft das Dienstprogramm `mpstat(1M)` diese Funktion wiederholt auf, um die neuesten Zählerwerte zu prüfen. Führen Sie `mpstat 1` in Ihrer Shell aus und beobachten Sie die Ausgabe. Brechen Sie `mpstat` nach einigen Sekunden durch Drücken von Strg-C in der Shell ab. Zum Beobachten der Zählerprüfung möchten wir einen Prüfpunkt aktivieren, der mit jedem Aufruf der Funktion `kstat_data_lookup(3KSTAT)` in `libkstat` durch den Befehl `mpstat` ausgelöst wird. Hierfür werden wir einen neuen DTrace-Provider einsetzen: `pid`. Der Provider `pid` ermöglicht die dynamische Erzeugung von Prüfpunkten in Benutzerprozessen an C-Symbolpositionen wie beispielsweise Eintrittspunkten. Mit Prüfpunktbeschreibungen in der folgenden Form können Sie den Provider `pid` dazu auffordern, zum Eintritt in eine und Rückkehr aus einer Benutzerfunktion einen Prüfpunkt zu erzeugen:

`pid`*Prozess-ID*`:`*Objektname*`:`*Funktionsname*`:entry`

`pid`*Prozess-ID*`:` *Objektname*`:`*Funktionsname* `:return`

Wenn Sie beispielsweise einen Prüfpunkt in dem Prozess mit der ID erzeugen möchten, der beim Eintritt in `kstat_data_lookup(3KSTAT)` ausgelöst wird, formulieren Sie die Prüfpunktbeschreibung wie folgt:

```
pid12345:libkstat:kstat_data_lookup:entry
```

Der Provider `pid` fügt an der der Prüfpunktbeschreibung entsprechenden Programmposition eine dynamische Instrumentation in den angegebenen Benutzerprozess ein. Die Prüfpunktimplementierung „lockt“ jeden Benutzer-Thread, der die instrumentierte Programmposition erreicht, in den Betriebssystemkernel und erzwingt dessen Eintritt in DTrace, wodurch der entsprechende Prüfpunkt ausgelöst wird. Obwohl also die Instrumentationsposition mit einem Benutzerprozess verbunden ist, werden die von Ihnen angegebenen DTrace-Prädikate und -Aktionen weiterhin im Kontext des Betriebssystemkernels ausgeführt. Der Provider `pid` wird in Kapitel 30, „Der Provider `pid`“ ausführlicher behandelt.

Damit Sie einen D-Programmquellcode nicht jedes Mal bearbeiten müssen, wenn Sie das Programm auf einen anderen Prozess anwenden möchten, können Sie so genannte *Makrovariablen* in Ihr Programm einsetzen, die zur Kompilierungszeit des Programms ausgewertet und durch die zusätzlichen `dtrace`-Befehlszeilenargumente ersetzt werden. Makrovariablen werden mit dem Dollarzeichen `$`, gefolgt von einem Bezeichner oder einer Ziffer angegeben. Wenn Sie den Befehl `dtrace -s` *Skript* `foo bar baz` ausführen, definiert der D-Compiler die Makrovariablen `$1`, `$2` und `$3` automatisch als die Symbole `foo`, `bar` bzw. `baz`.

Makrovariablen können in D-Programmausdrücken oder in Prüfpunktbeschreibungen benutzt werden. Die folgenden Prüfpunktbeschreibungen instrumentieren beispielsweise jede Prozess-ID, die als zusätzliches Argument für `dtrace` angegeben wird:

```
pid$1:libkstat:kstat_data_lookup:entry
{
    self->ksname = arg1;
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 != NULL/
{
    this->ksp = (kstat_named_t *)copyin(arg1, sizeof (kstat_named_t));
    printf("%s has ui64 value %u\n", copyinstr(self->ksname),
        this->ksp->value.ui64);
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 == NULL/
{
    self->ksname = NULL;
}
```

Makrovariablen und wieder verwendbare Skripten werden in Kapitel 15, „Scripting“ ausführlicher besprochen. Nachdem wir jetzt wissen, wie Benutzerprozesse anhand ihrer Prozess-ID instrumentiert werden, kehren wir zum Prüfen von Unionen zurück. Geben Sie den Quellcode für unser vollständiges Beispiel in einen Texteditor ein und speichern Sie ihn unter dem Namen `kstat.d`:

BEISPIEL 7–3 `kstat.d`: Ablaufverfolgungsaufrufe von `kstat_data_lookup`(3KSTAT)

```
pid$1:libkstat:kstat_data_lookup:entry
{
    self->ksname = arg1;
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 != NULL/
{
    this->ksp = (kstat_named_t *) copyin(arg1, sizeof (kstat_named_t));
    printf("%s has ui64 value %u\n",
        copyinstr(self->ksname), this->ksp->value.ui64);
}

pid$1:libkstat:kstat_data_lookup:return
/self->ksname != NULL && arg1 == NULL/
{
    self->ksname = NULL;
```

BEISPIEL 7–3 `kstat.d`: Ablaufverfolgungsaufrufe von `kstat_data_lookup`(3KSTAT) *(Fortsetzung)*

```
}
```

Führen Sie jetzt in einer Shell den Befehl `mpstat 1` aus, um `mpstat`(1M) in einem Modus zu starten, in dem statistische Daten gesammelt und einmal pro Sekunde gemeldet werden. Sobald `mpstat` läuft, führen Sie in der anderen Shell den Befehl `dtrace -q -s kstat.d ‘pgrep mpstat‘` aus. Sie sehen eine den Statistiken, auf die zugegriffen wird, entsprechende Ausgabe. Drücken Sie Strg-C, um `dtrace` abzubrechen und zur Shell-Eingabeaufforderung zurückzukehren.

```
# dtrace -q -s kstat.d ‘pgrep mpstat‘
cpu_ticks_idle has ui64 value 41154176
cpu_ticks_user has ui64 value 1137
cpu_ticks_kernel has ui64 value 12310
cpu_ticks_wait has ui64 value 903
hat_fault has ui64 value 0
as_fault has ui64 value 48053
maj_fault has ui64 value 1144
xcalls has ui64 value 123832170
intr has ui64 value 165264090
intrthread has ui64 value 124094974
pswitch has ui64 value 840625
inv_swtch has ui64 value 1484
cpumigrate has ui64 value 36284
mutex_adenters has ui64 value 35574
rw_rdfails has ui64 value 2
rw_wrfails has ui64 value 2
...
^C
#
```

Wenn Sie die Ausgabe in jedem Terminalfenster erfassen und jeden Wert von dem durch die vorige Iteration gemeldeten Wert in der Statistik subtrahieren, sollte es möglich sein, eine Beziehung zwischen der `dtrace`- und der `mpstat`-Ausgabe zu erkennen. Das Beispielprogramm zeichnet den Zählernamenzeiger beim Eintritt in die Lookup-Funktion auf und führt einen Großteil der Ablaufverfolgung nach der Rückkehr aus `kstat_data_lookup`(3KSTAT) durch. Wenn `arg1()` (der Rückgabewert) nicht `NULL()` ist, kopieren die in D integrierten Funktionen `copyinstr` und `copyin` die Funktionsresultate aus dem Benutzerprozess zurück nach DTrace. Nach dem Kopieren der `kstat`-Daten meldet das Beispielprogramm den `ui64`-Zählerwert aus der Union. In diesem vereinfachten Beispiel wird davon ausgegangen, dass `mpstat` Zähler prüft, die die Alternative `value.ui64` ansprechen. Versuchen Sie nun übungshalber, `kstat.d` mehrere Prädikate ansprechen und die der Komponente `data_type` entsprechende Unionsalternative ausgeben zu lassen. Sie können auch eine Version von `kstat.d` schreiben, die die Differenz zwischen aufeinander folgenden Datenwerten berechnet und eine ähnliche Ausgabe wie `mpstat` erzeugt.

Komponentengrößen und Versatz

Mit dem Operator `sizeof` lässt sich die Größe in Byte aller beliebigen D-Typen oder -Ausdrücke sowie von Strukturen oder Unionen ermitteln. Der Operator `sizeof` kann entweder auf einen Ausdruck oder auf den Namen eines in Klammern stehenden Typs angewendet werden:

`sizeof` *Ausdruck* `sizeof (`*Typname*`)`

So würde beispielsweise der Ausdruck `sizeof (uint64_t)` den Wert 8 zurückgeben, und der Ausdruck `sizeof (callinfo.ts)` würde im Quellcode des obigen Programmbeispiels ebenfalls 8 zurückgeben. Der formale Rückgabetyp des Operators `sizeof` ist der Typ-Aliasname `size_t`, der laut Definition eine vorzeichenlose Ganzzahl derselben Größe eines Zeigers im aktuellen Datenmodell ist und zur Darstellung der Byte-Anzahl verwendet wird. Bei Anwendung des Operators `sizeof` auf einen Ausdruck wird der Ausdruck vom D-Compiler überprüft, doch die resultierende Objektgröße wird zur Kompilierungszeit berechnet und es erfolgt keine Generierung von Code für den Ausdruck. Sie können `sizeof` überall dort einsetzen, wo eine ganzzahlige Konstante erforderlich ist.

Mit dem Operator `offsetof` lässt sich der Versatz (in Byte) einer Struktur- oder Unionskomponente zum Anfang des einem beliebigen Objekt des Typs struct oder union zugewiesenen Speicherbereichs ermitteln. Der Operator `offsetof` wird in Ausdrücken der folgenden Form verwendet:

`offsetof (`*Typname*`, `*Komponentenname*`)`

Dabei ist *Typname* der Name eines Struktur- oder Union-Typs oder ein Typ-Aliasname, und *Komponentenname* ist der Bezeichner einer Komponente dieser Struktur bzw. einer Alternative dieser Union. Genau wie `sizeof` gibt auch `offsetof` den Typ `size_t` zurück und kann überall dort in einem D-Programm eingesetzt werden, wo eine ganzzahlige Konstante zulässig ist.

Bit-Felder

D erlaubt auch die Definition von ganzzahligen Strukturkomponenten und Unionsalternativen mit beliebiger Anzahl Bits. Diese werden als *Bit-Felder* bezeichnet. Zur Deklaration eines Bit-Felds werden ein vorzeichenbehafteter oder vorzeichenloser Integer-Basistyp, ein Komponentenname und die dem Feld zuweisende Anzahl Bits angegeben:

```
struct s {
    int a : 1;
    int b : 3;
    int c : 12;
};
```

Die Breite des Bit-Felds ist eine ganzzahlige Konstante, die vom Komponentennamen durch einen angehängten Strichpunkt getrennt ist. Die Bit-Feldbreite muss positiv sein und ihre Bitanzahl darf die Breite des entsprechenden Integer-Basistyps nicht überschreiten. Bit-Felder mit einer Größe von über 64 Bit dürfen in D nicht deklariert werden. Bit-Felder in D bieten Kompatibilität mit der entsprechenden ANSI-C-Fähigkeit und Zugriff auf diese. In der Regel werden Bit-Felder zum Sparen von Speicherplatz eingesetzt oder dann, wenn das Strukturlayout mit dem Layout eines Hardwareregisters übereinstimmen muss.

Ein Bit-Feld ist ein Compiler-Konstrukt, mit dem sich die Anordnung einer Ganzzahl und einer Gruppe von Masken zum Extrahieren der Komponentenwerte automatisieren lässt. Dasselbe Ergebnis erhalten Sie, indem Sie die Masken einfach selbst definieren und den Operator & verwenden. Der C- und der D-Compiler versuchen stets, Bits so effizient wie möglich zusammenzupacken. Sie haben hierfür aber keine Vorgabe hinsichtlich der Reihenfolge oder Art und Weise. Deshalb ergeben identische Bit-Felder auf unterschiedlichen Compilern oder Architekturen nicht unbedingt dieselbe Bit-Anordnung. Wenn Sie ein stabiles Bit-Layout benötigen, sollten Sie die Bit-Masken selbst konstruieren und die Werte mit dem Operator & extrahieren.

Der Zugriff auf eine Komponente eines Bit-Felds erfolgt einfach durch die Angabe seines Namens in Kombination mit einem der Operatoren „.“ oder `->`, wie bei jeder anderen Strukturkomponente oder Unionsalternative. Das Bit-Feld wird automatisch auf den nächstgrößeren Integer-Typ zur Verwendung in einem beliebigen Ausdruck erweitert. Da der Speicherbereich für ein Bit-Feld nicht an einer Byte-Grenze ausgerichtet sein oder eine gerade Bytezahl als Größe aufweisen darf, können die Operatoren `sizeof` und `offsetof` nicht auf Komponenten eines Bit-Felds angewendet werden. Der D-Compiler verbietet außerdem das Abrufen der Adresse einer Bit-Feldkomponente mit dem Operator &.

◆ ◆ ◆ KAPITEL 8

Typ- und Konstantendefinitionen

In diesem Kapitel erfahren Sie, wie Typ-Aliasnamen und benannte Konstanten in D deklariert werden. Darüber hinaus wird in diesem Kapitel die Typ- und Namensraumverwaltung in D für Programm- und Betriebssystemtypen sowie -Bezeichner besprochen.

Typedef

Das Schlüsselwort `typedef` dient zum Deklarieren eines Bezeichners als Aliasname für einen bereits vorhandenen Typ. Wie bei allen D-Typ-Deklarationen wird das Schlüsselwort `typedef` außerhalb der Prüfpunktklauseln gesetzt. Die Deklarationen haben die folgende Form:

`typedef` *vorhandener_Typ neuer_Typ* `;`

dabei ist *vorhandener_Typ* eine beliebige Typ-Deklaration und *neuer_Typ* ist ein Bezeichner, der als Aliasname für diesen Typ verwendet werden soll. So wird beispielsweise die Deklaration:

```
typedef unsigned char uint8_t;
```

intern vom D-Compiler für die Erzeugung des Typ-Aliasnamens `uint8_t` verwendet. Typ-Aliasnamen können überall dort eingesetzt werden, wo ein normaler Typ, wie zum Beispiel der Typ eines Variablen- oder assoziativen Vektorwerts oder einer Tupel-Komponente, verwendet werden kann. Außerdem lässt sich `typedef` mit komplexeren Deklarationen wie der Definition einer neuen Struktur (`struct`) kombinieren:

```
typedef struct foo {
    int x;
    int y;
} foo_t;
```

In diesem Beispiel wird `struct foo` als derselbe Typ seines Aliasnamens, `foo_t`, definiert. In den C-System-Headern von Solaris wird ein `typedef`-Aliasname häufig durch den Zusatz `_t` gekennzeichnet.

Aufzählungen

Indem Sie symbolische Namen für Konstanten definieren, verbessern Sie nicht nur die Lesbarkeit der Programme, sondern erleichtern auch die zukünftige Programmpflege. Ein mögliches Verfahren besteht in der Definition einer *Aufzählung*, die eine Menge von Ganzzahlen einem Satz Bezeichnern, den so genannten Aufzählungskonstanten, zuordnen. Diese erkennt der Compiler und ersetzt sie durch die entsprechenden ganzzahligen Werte. Aufzählungen werden mit Deklarationen wie der folgenden definiert:

```
enum colors {
    RED,
    GREEN,
    BLUE
};
```

Die erste Aufzählungskonstante in der Aufzählung, `ROT`, erhält den Wert Null, und jeder nachfolgende Bezeichner den nächsthöheren ganzzahligen Wert. Sie können für jede beliebige Aufzählungskonstante auch einen expliziten ganzzahligen Wert angeben, indem Sie ihr ein Gleichheitszeichen und eine ganzzahlige Konstante nachstellen:

```
enum colors {
    RED = 7,
    GREEN = 9,
    BLUE
};
```

Der Aufzählungskonstante `BLAU` wird vom Compiler der Wert 10 zugewiesen, da für sie kein Wert angegeben ist und die vorhergehende Aufzählungskonstante den Wert 9 besitzt. Nachdem eine Aufzählung definiert wurde, können deren Auszählungskonstanten überall dort in einem D-Programm verwendet werden, wo ganzzahlige Konstanten zum Einsatz kommen. Die Aufzählung `enum colors` ist außerdem als Typ `int` definiert. Der D-Compiler lässt Variablen des Typs `enum` überall dort zu, wo ein `int` verwendet werden kann. Einer Variable des Typs `enum` darf jeder ganzzahlige Wert zugewiesen werden. Wenn der Typname nicht erforderlich ist, können Sie in der Deklaration auf den Namen `enum` verzichten.

Aufzählungskonstanten sind in allen nachfolgenden Klauseln und Deklarationen eines Programms sichtbar. Deshalb kann ein Name für eine Aufzählungskonstante nicht in mehreren Aufzählungen definiert werden. Sie können allerdings sowohl in derselben als auch in verschiedenen Aufzählungen mehrere Aufzählungskonstanten mit demselben Wert definieren. Es ist auch möglich, einer Variable des Aufzählungstyps Ganzzahlen ohne entsprechende Aufzählungskonstante zuzuweisen.

Die Syntax für D-Aufzählungen ist identisch mit der entsprechenden Syntax in ANSI-C. D ermöglicht auch den Zugriff auf Aufzählungen, die im Betriebssystemkern und dessen ladbaren Modulen definiert sind. Diese Aufzählungskonstanten sind im D-Programm jedoch nicht global sichtbar. Kernel-Aufzählungskonstanten sind nur dann sichtbar, wenn sie zum Vergleich

mit einem Objekt des entsprechenden Aufzählungstyps als Argument für einen binären Vergleichsoperator eingesetzt werden. So besitzt beispielsweise die Funktion uiomove(9F) einen Parameter des Typs enum uio_rw, der wie folgt definiert ist:

```
enum uio_rw { UIO_READ, UIO_WRITE };
```

Die Aufzählungskonstanten UIO_READ und UIO_WRITE sind in einem D-Programm normalerweise nicht sichtbar, können aber durch Vergleich mit einem Wert des Typs enum uio_rw auf globale Sichtbarkeit erweitert werden. Dies zeigt die nächste Beispielklausel:

```
fbt::uiomove:entry
/args[2] == UIO_WRITE/
{
    ...
}
```

In diesem Beispiel werden Aufrufe der Funktion uiomove(9F) für Schreibanforderungen verfolgt. Dabei wird args[2], eine Variable des Typs enum uio_rw, mit der Aufzählungskonstante UIO_WRITE verglichen. Da das Argument auf der linken Seite den Aufzählungstyp besitzt, durchsucht der D-Compiler die Aufzählung in dem Versuch, den Bezeichner auf der rechten Seite aufzulösen. Diese Einrichtung schützt Ihre D-Programme vor Namenskonflikten mit der umfangreichen Sammlung der im Betriebssystemkernel definierten Aufzählungen.

Inline

Benannte Konstanten in D können auch anhand von inline-Direktiven definiert werden. Dabei handelt es sich um ein allgemeineres Hilfsmittel für die Erzeugung von Namen, die bei der Kompilierung durch vordefinierte Werte oder Ausdrücke ersetzt werden. Inline-Direktiven stellen im Vergleich zu der #define-Direktive des C-Preprozessors eine leistungsfähigere lexikalische Ersetzung zur Verfügung, da der Ersetzung ein tatsächlicher Typ zugewiesen ist und sie nicht nur anhand einer Menge lexikalischer Symbole (Token), sondern unter Verwendung des kompilierten Syntaxbaums durchgeführt wird. Inline-Direktiven werden in folgender Form deklariert:

```
inline Typ Name = Ausdruck ;
```

wobei *Typ* eine Typdeklaration eines vorhandenen Typs ist, *Name* ein beliebiger gültiger D-Bezeichner, der noch nicht als Inline- oder globale Variable definiert wurde, und *Ausdruck* ein beliebiger gültiger D-Ausdruck. Sobald die Inline-Direktive verarbeitet ist, setzt der D-Compiler die kompilierte Form von *Ausdruck* für jede nachfolgende Instanz von *Name* im Programmquellcode ein. Das nächste D-Programm verfolgt beispielsweise die Zeichenkette "hello" und den ganzzahligen Wert 123:

```
inline string hello = "hello";
inline int number = 100 + 23;

BEGIN
{
    trace(hello);
    trace(number);
}
```

Ein Inline-Name ist überall dort zulässig, wo eine globale Variable des entsprechenden Typs verwendet werden kann. Wenn die Inline-Direktive zur Kompilierungszeit eine ganzzahlige oder eine Zeichenkettenkonstante ergibt, kann der Inline-Name auch in einem Kontext verwendet werden, in dem konstante Ausdrücke wie zum Beispiel skalare Vektordimensionen erforderlich sind.

Zur Auswertung der Anweisung gehört die Prüfung der Inline-Direktive auf Syntaxfehler. Der Ergebnistyp des Ausdrucks muss mit dem in der Inline-Direktive definierten Typ vereinbar sein. Hierbei gelten dieselben Regeln wie für den D-Zuweisungsoperator (=). Ein Inline-Ausdruck darf nicht auf den Inline-Namen selbst verweisen: Rekursive Definitionen sind nicht erlaubt.

Mit der DTrace-Software werden im Systemverzeichnis `/usr/lib/dtrace` eine Reihe von D-Quelldateien installiert, die Inline-Direktiven zur Verwendung in Ihren D-Programmen enthalten. So enthält beispielsweise die Bibliothek `signal.d` Direktiven der Form:

```
inline int SIGHUP = 1;
inline int SIGINT = 2;
inline int SIGQUIT = 3;
...
```

Diese Inline-Definitionen ermöglichen den Zugriff auf den in `signal(3HEAD)` beschriebenen aktuellen Signalnamensatz in Solaris. Analog enthält die Bibliothek `errno.d` Inline-Direktiven für die `errno`-Konstanten in C, die in `Intro(2)` beschrieben sind.

Standardmäßig nimmt der D-Compiler alle bereitgestellten D-Bibliotheksdateien automatisch auf, sodass die Definitionen in jedem D-Programm verwendet werden können.

Namensräume für Typen

Dieser Abschnitt befasst sich mit Namensräumen in D und Problemen in Bezug auf Namensräume für Typen. In herkömmlichen Sprachen wie ANSI-C ergibt sich die Typ-Sichtbarkeit daraus, ob ein Typ in eine Funktion oder eine andere Deklaration eingebettet ist. Im externen Gültigkeitsbereich eines C-Programms deklarierte Typen haben einen einzelnen, globalen Namensraum und sind im gesamten Programm sichtbar. Die in C-Definitionsdateien vereinbarten Typen werden in der Regel in diesen externen

Gültigkeitsbereich aufgenommen. Im Gegensatz zu diesen Sprachen ermöglicht D den Zugriff auf Typen von mehreren externen Gültigkeitsbereichen.

Die Sprache D begünstigt die dynamische Beobachtungsfähigkeit über mehrere Ebenen eines Software-Stacks hinweg, einschließlich des Betriebssystemkernels, einer Gruppe ladbarer Kernelmodule und der auf dem System laufenden Benutzerprozesse. Ein einzelnes D-Programm kann Prüfpunkte zum Abrufen von Daten aus mehreren Kernelmodulen oder anderen Softwareeinheiten instanziieren, die in unabhängige binäre Objekte kompiliert werden. Aus diesem Grund ist es denkbar, dass in dem Universum der DTrace und dem D-Compiler zur Verfügung stehenden Typen mehrere Datentypen mit demselben Namen, möglicherweise mit unterschiedlichen Definitionen, vorliegen. Um dieser Situation Herr zu werden, verbindet der D-Compiler mit jedem Typ einen Namensraum, der in dem ihn enthaltenden Programmobjekt vereinbart ist. Typen aus einem bestimmten Programmobjekt können durch Angabe des Objektnamens und des Backquote-Operators (‘) in einem beliebigen Typnamen angesprochen werden.

Wenn beispielsweise ein Kernelmodul namens `foo` die folgende C-Typdeklaration enthält:

```
typedef struct bar {
    int x;
} bar_t;
```

dann kann auf die Typen `struct bar` und `bar_t` aus D anhand der folgenden Typnamen zugegriffen werden:

```
struct foo‘bar                foo‘bar_t
```

Der Backquote-Operator kann in jedem Kontext verwendet werden, in dem ein Typname geeignet ist, einschließlich bei der Angabe des Typs für D-Variablendeklarationen oder ausdrückliche Typumwandlungen in D-Prüfpunktklauseln.

Der D-Compiler stellt auch zwei spezielle Namensräume für integrierte Typen, `C` und `D`, zur Verfügung. Der Namensraum für C-Typen wird anfänglich mit den ANSI-C-Standardtypen wie zum Beispiel `int` gefüllt. Darüber hinaus werden mit dem C-Preprozessor cpp(1) und der `dtrace`-Option `-C` erfasste Typdefinitionen von dem C-Gültigkeitsbereich verarbeitet und in ihn aufgenommen. Folglich können Sie C-Definitionsdateien aufnehmen, die bereits in einem anderen Typ-Namensraum sichtbare Typdeklarationen enthalten, ohne dadurch Kompilierungsfehler zu verursachen.

Der Namensraum für D-Typen wird anfänglich mit den D-internen Typen wie `int` und `string` und Typ-Aliasnamen wie `uint32_t` gefüllt. Neue Typdeklarationen, die im D-Programmcode vorkommen, werden automatisch dem Namensraum für D-Typen hinzugefügt. Wenn Sie in einem D-Programm einen komplexen Typ wie `struct` erzeugen, dessen Komponententypen aus anderen Namensräumen stammen, werden die Komponententypen durch die Deklaration in den D-Namensraum kopiert.

Sollte der D-Compiler auf eine Typdeklaration treffen, die keine explizite Namensraumangabe mit dem Backquote-Operator enthält, durchsucht der Compiler die Menge der aktiven Typ-Namensräume nach einer Entsprechung für den angegebenen Typnamen. Der C-Namensraum wird stets zuerst durchsucht. Danach erfolgt die Suche im D-Namensraum. Kann der Typname weder im C- noch im D-Namensraum gefunden werden, werden die Typ-Namensräume der aktiven Kernelmodule in aufsteigender Reihenfolge nach Kernelmodul-ID durchsucht. Diese Reihenfolge gewährleistet, dass die den inneren Kern bildenden binären Objekte vor etwaigen ladbaren Kernelmodulen durchsucht werden, garantiert aber keinerlei Reihenfolgeneigenschaften über die ladbaren Module hinaus. Benutzen Sie den Backquote-Operator für den Zugriff auf Typen, die in ladbaren Kernelmodulen definiert sind, um Typnamenskonflikte mit anderen Kernelmodulen zu vermeiden.

Für den automatischen Zugriff auf die Typen des Betriebssystemquellcodes stützt sich der D-Compiler auf komprimierte ANSI-C-Debugging-Informationen aus den zentralen Solaris-Kernelmodulen, sodass nicht auf die entsprechenden C-Include-Dateien zugegriffen werden muss. Diese symbolischen Debugging-Informationen stehen möglicherweise nicht für alle Kernelmodule auf dem System zur Verfügung. Bei dem Versuch, auf einen Typ im Namensraum eines Moduls ohne die für DTrace vorgesehenen komprimierten C-Debugging-Informationen zuzugreifen, gibt der D-Compiler einen Fehler aus.

◆ ◆ ◆ KAPITEL 9

9 Aggregate

Für die Instrumentierung des Systems zur Beantwortung leistungsbezogener Fragestellungen bietet es sich an, zu überlegen, wie Daten zu einer Antwort auf eine bestimmte Frage zusammengefasst werden können, anstatt die Daten von mehreren verschiedenen Prüfpunkten erfassen zu lassen. Wenn beispielsweise die Anzahl der von einer Benutzer-ID durchgeführten Systemaufrufe festgestellt werden soll, sind die bei den *einzelnen* Systemaufrufen erfassten Daten wahrscheinlich weniger interessant. Wir möchten einfach nur eine Tabelle mit Benutzer-IDs und Systemaufrufen sehen. Traditionsgemäß würde man zum Beantworten dieser Frage bei jedem Systemaufruf Informationen sammeln und die Informationen mit einem Tool wie `awk(1)` oder `perl(1)` nachträglich verarbeiten. In DTrace ist die Zusammenfassung von Daten jedoch eine Operation der ersten Klasse. Dieses Kapitel befasst sich mit den DTrace-Einrichtungen für die Manipulation von *Aggregaten*.

Aggregatfunktionen

Eine *Aggregatfunktion* ist eine Funktion mit den folgenden Eigenschaften:

$$f(f(x_0) \cup f(x_1) \cup \ldots \cup f(x_n)) = f(x_0 \cup x_1 \cup \ldots \cup x_n)$$

wobei x_n eine Menge beliebiger Daten ist. Das bedeutet, dass die Anwendung einer Aggregatfunktion auf Untergruppen des Ganzen und anschließend auf die Ergebnisse dasselbe Resultat erbringt, wie die Anwendung der Funktion auf das Ganze. Betrachten wir zum Beispiel eine Funktion `SUM`, die die Summe eines gegebenen Datensatzes ergibt. Wenn die unverarbeiteten Daten aus {2, 1, 2, 5, 4, 3, 6, 4, 2} bestehen, dann führt die Anwendung von `SUM` auf den gesamten Datensatz zum Ergebnis {29}. Analog führt die Anwendung von `SUM` auf die aus den ersten drei Elementen bestehende Untergruppe zum·Ergebnis \{5\}. Wenn wir `SUM` auf die drei nachfolgenden Elemente anwenden, erhalten wir \{12\}, und die Anwendung von `SUM` auf die übrigen drei Elemente ergibt ebenso \{12}. `SUM` ist eine Aggregatfunktion, da wir durch ihre Anwendung auf die Menge der Ergebnisse, {5, 12, 12}, dasselbe Ergebnis erhalten wie bei Anwendung von `SUM` auf die ursprünglichen Daten, nämlich \{29\}.

Nicht alle Funktionen haben diese zusammenfassende Wirkung. Ein Beispiel für eine Nicht-Aggregatfunktion ist die Funktion `MEDIAN`, die das mittlere Element eines Datensatzes ermittelt. (Ein Element, für das in einem Datensatz gleich viele größere wie kleinere Elemente vorhanden sind, ist das mittlere Element im Datensatz.) Der `MEDIAN` wird durch Sortieren des Datensatzes und Auswahl des mittleren Elements abgeleitet. Kehren wir zu den ursprünglichen unverarbeiteten Daten zurück und wenden wir `MEDIAN` auf die Untergruppe der ersten drei Elemente an. Das Ergebnis ist {2}. (Der sortierte Satz lautet {1, 2, 2}; {2} ist der aus dem mittleren Element bestehende Satz.) Analog ergibt die Anwendung von `MEDIAN` auf die nächsten drei Elemente \{4\} und die Anwendung von `MEDIAN` auf die letzten drei Elemente ebenfalls \{4}. Wenn wir `MEDIAN` auf die einzelnen Untergruppen anwenden, erhalten wir also den Datensatz {2, 4, 4}. Wenden wir `MEDIAN` nun auf diesen Satz an, beträgt das Ergebnis {4}. Wenn wir aber den ursprünglichen Satz sortieren, erhalten wir {1, 2, 2, 2, 3, 4, 4, 5, 6}. Das Ergebnis der Anwendung von `MEDIAN` auf diesen Satz beträgt {3}. Da diese beiden Ergebnisse nicht übereinstimmen, ist `MEDIAN` keine Aggregatfunktion.

Viele gebräuchliche Funktionen für die Analyse von Datensätzen sind Aggregatfunktionen. Dazu gehören Funktionen zum Zählen der Elemente im Datensatz, zum Berechnen des niedrigsten oder des höchsten Werts im Datensatz und zum Addieren aller Elemente im Datensatz. Die Bestimmung des arithmetischen Mittels einer Menge lässt sich aus der Funktion zum Zählen der Elemente und jener zum Berechnen der Summe aller Elemente im Datensatz kombinieren.

Verschiedene nützliche Funktionen sind jedoch keine Aggregatfunktionen. Hierzu gehören Funktionen zum Berechnen des Modus (des häufigsten Elements), des Mittelwerts oder der Standardabweichung einer Menge.

Die Anwendung von Aggregatfunktionen auf Daten während der Ablaufverfolgung bringt eine Reihe von Vorteilen mit:

- Es muss nicht der gesamte Datensatz gespeichert werden. Jedes Mal, wenn dem Satz ein neues Element hinzugefügt werden soll, wird die Aggregatfunktion auf den aus dem aktuellen Zwischenergebnis bestehenden Datensatz und das neue Element angewendet. Nach der Berechnung des neuen Ergebnisses kann das neue Element verworfen werden. Durch diesen Prozess lässt sich die erforderliche Speicherkapazität um den Faktor der Anzahl Datenpunkte verringern, die häufig sehr hoch ist.
- Die Datensammlung verursacht keine schädlichen Skalierbarkeitsprobleme. Aggregatfunktionen ermöglichen die Aufbewahrung von Zwischenergebnissen *per CPU* anstatt in einer gemeinsamen Datenstruktur. DTrace wendet dann die Aggregatfunktion auf die Menge aus den Zwischenergebnissen per CPU an, um das systemweite Endergebnis zu produzieren.

Aggregate

DTrace speichert die Ergebnisse von Aggregatfunktionen in einem Objekt namens *Aggregat*. Die Aggregatergebnisse werden ähnlich wie bei assoziativen Vektoren mit einem Tupel von Ausdrücken indiziert. Die Syntax für Aggregate in D lautet:

```
@Name[ Schlüssel ] = Aggfunk ( Args );
```

wobei *Name* für den Namen des Aggregats, *Schlüssel* für eine Liste mit Komma getrennter D-Ausdrücke, *Aggfunkt* für eine der DTrace-Aggregatfunktionen und *Argumente* für eine Liste mit Komma getrennter Argumente für die Aggregatfunktion stehen. Der Aggregat*name* ist ein D-Bezeichner mit dem vorangestellten Sonderzeichen `@`. Alle in Ihren D-Programmen benannten Aggregate sind globale Variablen. Es gibt keine thread- oder klausel-lokalen Aggregate. Die Aggregatnamen werden, getrennt von anderen globalen D-Variablen, in einem separaten Bezeichner-Namensraum geführt. Denken Sie bei der Wiederverwendung von Namen daran, dass `a` und `@a` nicht dieselbe Variable sind. Der spezielle Aggregatname `@` kann in einfachen D-Programmen zum Benennen anonymer Aggregate eingesetzt werden. Der D-Compiler behandelt den Namen als Alias für den Aggregatnamen `@_`.

Die folgende Tabelle zeigt die Aggregatfunktionen von DTrace. Die meisten Aggregatfunktionen nehmen nur ein einziges Argument an, das die neue Information darstellt.

TABELLE 9–1 Aggregatfunktionen in DTrace

Funktionsname	Argumente	Ergebnis
`count`	none	Anzahl der Aufrufe.
`sum`	Skalarer Ausdruck	Gesamtwert der angegebenen Ausdrücke.
`avg`	Skalarer Ausdruck	Arithmetisches Mittel der angegebenen Ausdrücke.
`min`	Skalarer Ausdruck	Kleinster Wert in den angegebenen Ausdrücken.
`max`	Skalarer Ausdruck	Größter Wert in den angegebenen Ausdrücken.
`lquantize`	Skalarer Ausdruck, Untergrenze, Obergrenze, Schrittwert	Lineare Häufigkeitsverteilung der Werte der angegebenen Ausdrücke, beschränkt durch den angegebenen Bereich. Inkrementiert den Wert in der *höchsten* Menge, die *kleiner* als der angegebene Ausdruck ist.
`quantize`	Skalarer Ausdruck	Eine Quadrat-Häufigkeitsverteilung der Werte der angegebenen Ausdrücke. Inkrementiert den Wert in der *höchsten* Quadratmenge, die *kleiner* als der angegebene Ausdruck ist.

Wenn beispielsweise die Anzahl der `write(2)`-Systemaufrufe im System gezählt werden sollen, könnten Sie eine informative Zeichenkette als Schlüssel und die Aggregatfunktion `count()` verwenden:

```
syscall::write:entry
{
    @counts["write system calls"] = count();
}
```

Der Befehl `dtrace` gibt die Aggregatergebnisse standardmäßig nach Abschluss des Prozesses aus, entweder als Resultat einer expliziten `END`-Aktion oder auf die Betätigung von Strg-C durch den Benutzer hin. Die folgende Beispielausgabe zeigt das Ergebnis nach Ausführung dieses Befehls, einer Wartezeit von einigen Sekunden und der anschließenden Betätigung von Strg-C:

```
# dtrace -s writes.d
dtrace: script ’./writes.d’ matched 1 probe
^C

  write system calls                                              179
#
```

Mit der Variable `execname` als Schlüssel für ein Aggregat lassen sich die Systemaufrufe per Prozessnamen zählen:

```
syscall::write:entry
{
    @counts[execname] = count();
}
```

Die folgende Beispielausgabe zeigt das Ergebnis nach Ausführung dieses Befehls, einer Wartezeit von mehreren Sekunden und Drücken von Strg-C:

```
# dtrace -s writesbycmd.d
dtrace: script ’./writesbycmd.d’ matched 1 probe
^C

  dtrace                                                            1
  cat                                                               4
  sed                                                               9
  head                                                              9
  grep                                                             14
  find                                                             15
  tail                                                             25
  mountd                                                           28
  expr                                                             72
  sh                                                              291
  tee                                                             814
  def.dir.flp                                                    1996
  make.bin                                                       2010
#
```

Alternativ können die Schreibzugriffe nach Prozessnamen und Dateibezeichner zusammen organisiert werden. Der Dateibezeichner ist das erste Argument für `write(2)`. Im folgenden Beispiel wird also ein aus `execname` und `arg0` zusammengesetzter Schlüssel verwendet:

```
syscall::write:entry
{
    @counts[execname, arg0] = count();
}
```

Die Ausführung dieses Befehls ergibt eine Tabelle wie in folgendem Beispiel, mit Prozessnamen und Dateibezeichner:

```
# dtrace -s writesbycmdfd.d
dtrace: script ’./writesbycmdfd.d’ matched 1 probe
^C

  cat                                                       1      58
  sed                                                       1      60
  grep                                                      1      89
  tee                                                       1     156
  tee                                                       3     156
  make.bin                                                  5     164
  acomp                                                     1     263
  macrogen                                                  4     286
  cg                                                        1     397
  acomp                                                     3     736
  make.bin                                                  1     880
  iropt                                                     4    1731
#
```

Im nächsten Beispiel wird die durchschnittlich im write-Systemaufruf verbrachte Zeit je Prozess angezeigt. Dabei wird der Aggregatfunktion `avg()` der Ausdruck als Argument übergeben, aus dem der Durchschnitt berechnet werden soll. In diesem Beispiel wird der Durchschnitt der im Systemaufruf abgelaufenen Gesamtzeit berechnet:

```
syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
/self->ts/
{
    @time[execname] = avg(timestamp - self->ts);
    self->ts = 0;
}
```

Die folgende Beispielausgabe zeigt das Ergebnis nach Ausführung dieses Befehls, einer Wartezeit von mehreren Sekunden und Drücken von Strg-C:

```
# dtrace -s writetime.d
dtrace: script ’./writetime.d’ matched 2 probes
^C

  iropt                                                   31315
  acomp                                                   37037
  make.bin                                                63736
  tee                                                     68702
  date                                                    84020
  sh                                                      91632
  dtrace                                                 159200
  ctfmerge                                               321560
  install                                                343300
  mcs                                                    394400
  get                                                    413695
  ctfconvert                                             594400
  bringover                                             1332465
  tail                                                  1335260
#
```

Ein Durchschnittswert kann zwar nützlich sein, liefert aber häufig nicht genug Detailinformationen zum Verstehen der Verteilung von Datenpunkten. Um die Verteilung genauer zu verstehen, setzen wir wie folgt die Aggregatfunktion `quantize()` ein:

```
syscall::write:entry
{
    self->ts = timestamp;
}

syscall::write:return
/self->ts/
{
    @time[execname] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

Da aus jeder Ausgabezeile ein Häufigkeitsverteilungsdiagramm wird, fällt die Ausgabe dieses Skripts bedeutend länger aus als die vorherigen. Dieses Beispiel zeigt nur eine Auswahl der Beispielausgabe:

```
  lint
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |                                         2
           32768 |                                         0
```

```
          65536 |@@@@@@@@@@@@@@@@@@@                74
         131072 |@@@@@@@@@@@@@@@                    59
         262144 |@@@                                14
         524288 |                                   0

acomp
           value  ------------- Distribution ------------- count
            4096 |                                   0
            8192 |@@@@@@@@@@@@@                      840
           16384 |@@@@@@@@@@@@                       750
           32768 |@@                                 165
           65536 |@@@@@@                             460
          131072 |@@@@@@                             446
          262144 |                                   16
          524288 |                                   0
         1048576 |                                   1
         2097152 |                                   0

iropt
           value  ------------- Distribution ------------- count
            4096 |                                   0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@        4149
           16384 |@@@@@@@@@@                         1798
           32768 |@                                  332
           65536 |@                                  325
          131072 |@@                                 431
          262144 |                                   3
          524288 |                                   2
         1048576 |                                   1
         2097152 |                                   0
```

Beachten Sie, dass die Zeilen der Häufigkeitsverteilung *immer* Zweierpotenzen darstellen. Jede Zeile gibt die Anzahl der Elemente an, die *größer gleich* dem entsprechenden Wert, aber *kleiner als* der Wert der nächsthöheren Zeile sind. So zeigt beispielsweise die obige Ausgabe, dass `iropt` 4.149 Schreibzugriffe mit einer Dauer von 8.192 bis 16.383 (einschließlich) Nanosekunden hatte.

`quantize()` eignet sich für einen schnellen Einblick in die Daten, aber nicht so sehr, wenn Sie eine Verteilung über lineare Werte untersuchen müssen. Zum Anzeigen einer linearen Werteverteilung verwenden Sie die Aggregatfunktion `lquantize. ()` Die Funktion `lquantize()` nimmt zusätzlich zu einem D-Ausdruck drei Argumente an: eine Untergrenze, eine Obergrenze und eine Schrittweite. Zur Betrachtung der Verteilung der Schreibzugriffe nach Dateibezeichner wäre eine Zweierpotenz-Quantisierung beispielsweise nicht wirkungsvoll. Gehen Sie hierzu stattdessen mit einer linearen Quantisierung und kleinem Wertebereich vor, wie im nächsten Beispiel dargestellt:

```
syscall::write:entry
{
```

```
    @fds[execname] = lquantize(arg0, 0, 100, 1);
}
```

Wenn Sie dieses Skript einige Sekunden lang ausführen, erhalten Sie eine große Menge an Informationen. Dieses Beispiel zeigt einen Ausschnitt einer typischen Ausgabe:

```
  mountd
           value  ------------- Distribution ------------- count
              11 |                                         0
              12 |@                                        4
              13 |                                         0
              14 |@@@@@@@@@@@@@@@@@@@@@@@@@@               70
              15 |                                         0
              16 |@@@@@@@@@@@@                             34
              17 |                                         0

  xemacs-20.4
           value  ------------- Distribution ------------- count
               6 |                                         0
               7 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  521
               8 |                                         0
               9 |                                         1
              10 |                                         0

  make.bin
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  3596
               2 |                                         0
               3 |                                         0
               4 |                                         42
               5 |                                         50
               6 |                                         0

  acomp
           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@                                    1156
               2 |                                         0
               3 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    6635
               4 |@                                        297
               5 |                                         0

  iropt
           value  ------------- Distribution ------------- count
               2 |                                         0
               3 |                                         299
               4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  20144
               5 |                                         0
```

Die Funktion `lquantize()` ermöglicht außerdem die Zusammenfassung der Zeit ab einem bestimmten Punkt in der Vergangenheit. Mit dieser Technik können Sie Änderungen des Verhaltens im Verlauf der Zeit beobachten. Das folgende Beispiel zeigt die Änderung im Systemaufruf-Verhalten über die Lebensdauer eines Prozesses, der den Befehl `date(1)` ausführt:

```
syscall::exec:return,
syscall::exece:return
/execname == "date"/
{
    self->start = vtimestamp;
}

syscall:::entry
/self->start/
{
    /*
     * We linearly quantize on the current virtual time minus our
     * process's start time.  We divide by 1000 to yield microseconds
     * rather than nanoseconds.  The range runs from 0 to 10 milliseconds
     * in steps of 100 microseconds; we expect that no date(1) process
     * will take longer than 10 milliseconds to complete.
     */
    @a["system calls over time"] =
        lquantize((vtimestamp - self->start) / 1000, 0, 10000, 100);
}

syscall::rexit:entry
/self->start/
{
    self->start = 0;
}
```

Je mehr `date(1)`-Prozesse ausgeführt werden, desto tiefer der Einblick in das Systemaufruf-Verhalten, das das obige Skript bietet. Um das Ergebnis zu sehen, führen Sie `sh -c 'while true; do date >/dev/null; done'` in einem Fenster und das D-Skript in einem anderen aus. Das Skript erzeugt ein Profil des Systemaufruf-Verhaltens des Befehls `date(1)`:

```
# dtrace -s dateprof.d
dtrace: script './dateprof.d' matched 218 probes
^C

  system calls over time
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@                                       20530
             100 |@@@@@@                                   48814
             200 |@@@                                      28119
             300 |@                                        14646
```

```
 400 |@@@@@                                    41237
 500 |                                         1259
 600 |                                         218
 700 |                                         116
 800 |@                                        12783
 900 |@@@                                      28133
1000 |                                         7897
1100 |@                                        14065
1200 |@@@                                      27549
1300 |@@@                                      25715
1400 |@@@@                                     35011
1500 |@@                                       16734
1600 |                                         498
1700 |                                         256
1800 |                                         369
1900 |                                         404
2000 |                                         320
2100 |                                         555
2200 |                                         54
2300 |                                         17
2400 |                                         5
2500 |                                         1
2600 |                                         7
2700 |                                         0
```

Diese Ausgabe gibt einen groben Eindruck der verschiedenen Phasen des Befehls `date(1)` in Bezug auf die vom Kernel benötigten Dienste. Zum besseren Verständnis dieser Phasen kann es nützlich sein, festzustellen, welche Systemaufrufe wann stattfinden. Dafür könnten wir das D-Skript so abändern, dass die Aggregation an der Variable `probefunc` anstatt an einer konstanten Zeichenkette erfolgt.

Anzeige von Aggregaten

Standardmäßig werden Aggregate in der Reihenfolge ihres Auftretens im D-Programm angezeigt. Mit der Funktion `printa()` für die Ausgabe der Aggregate können Sie dieses Verhalten außer Kraft setzen. Die Funktion `printa()` bietet auch die Möglichkeit, die Aggregatdaten genau zu formatieren. Hierzu setzen Sie, wie in Kapitel 12, „Formatierung der Ausgabe“ beschrieben, eine Formatzeichenkette ein.

Wenn ein Aggregat nicht durch eine `printa()`-Anweisung im D-Programm formatiert wird, erstellt der Befehl `dtrace` eine Momentaufnahme der Aggregatdaten und gibt die Ergebnisse nach Abschluss der Ablaufverfolgung im Standard-Aggregatformat aus. Bei Verwendung einer `printa()`-Anweisung zur Formatierung eines bestimmten Aggregats wird das Standardverhalten deaktiviert. Indem Sie die Anweisung `printa(@`*Aggregatname*) in eine `dtrace:::END`-Prüfpunktklausel in Ihrem Programm einfügen, erhalten Sie äquivalente Ergebnisse. Im Standardausgabeformat für die Aggregatfunktionen `avg()`, `count()`, `min()`,

`max()` und `sum()` wird ein ganzzahliger Dezimalwert für den Aggregatwert jedes Tupels angezeigt. Das Standardausgabeformat der Aggregatfunktionen `lquantize()` und `quantize()` zeigt die Ergebnisse in Form einer ASCII-Tabelle. Aggregat-Tupel werden so ausgegeben, als hätte man `trace()` auf jedes Tupelelement angewendet.

Datennormalisierung

Beim Anhäufen von Daten über einen gewissen Zeitraum kann es hilfreich sein, die Daten in Bezug auf einen konstanten Faktor zu *normalisieren.* Dank dieser Technik lassen sich unzusammenhängende Daten leichter vergleichen. So sollen möglicherweise beim Aggregieren von Systemaufrufen die Systemaufrufe nicht als absoluter Wert über den Verlauf der Ausführung, sondern in Häufigkeit pro Sekunde angezeigt werden. Die DTrace-Aktion `normalize()` ermöglicht eine derartige Normalisierung der Daten. Die Parameter für `normalize()` sind ein Aggregat und ein Normalisierungsfaktor. Die Ausgabe der Aggregatfunktion zeigt jeden Wert dividiert durch den Normalisierungsfaktor.

Das folgende Beispiel verdeutlicht das Aggregieren von Daten nach Systemaufruf:

```
#pragma D option quiet

BEGIN
{
    /*
     * Get the start time, in nanoseconds.
     */
    start = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

END
{
    /*
     * Normalize the aggregation based on the number of seconds we have
     * been running.  (There are 1,000,000,000 nanoseconds in one second.)
     */
    normalize(@func, (timestamp - start) / 1000000000);
}
```

Wenn das obige Skript kurz ausgeführt wird, erhalten wir auf einem Desktoprechner die folgende Ausgabe:

```
# dtrace -s ./normalize.d
 ^C
```

```
  syslogd                                                   0
  rpc.rusersd                                               0
  utmpd                                                     0
  xbiff                                                     0
  in.routed                                                 1
  sendmail                                                  2
  echo                                                      2
  FvwmAuto                                                  2
  stty                                                      2
  cut                                                       2
  init                                                      2
  pt_chmod                                                  3
  picld                                                     3
  utmp_update                                               3
  httpd                                                     4
  xclock                                                    5
  basename                                                  6
  tput                                                      6
  sh                                                        7
  tr                                                        7
  arch                                                      9
  expr                                                     10
  uname                                                    11
  mibiisa                                                  15
  dirname                                                  18
  dtrace                                                   40
  ksh                                                      48
  java                                                     58
  xterm                                                   100
  nscd                                                    120
  fvwm2                                                   154
  prstat                                                  180
  perfbar                                                 188
  Xsun                                                   1309
  .netscape.bin                                          3005
```

`normalize()` legt den Normalisierungsfaktor für das angegebene Aggregat fest. Die zugrunde liegenden Daten werden dadurch jedoch nicht geändert. `denormalize()` übernimmt nur ein Aggregat. Wenn wir dem vorigen Beispiel die Denormalisierungsaktion hinzufügen, erhalten wir sowohl die unverarbeitete Anzahl der Systemaufrufe als auch die Frequenz pro Sekunde:

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
}
```

```
syscall:::entry
{
    @func[execname] = count();
}

END
{
    this->seconds = (timestamp - start) / 1000000000;
    printf("Ran for %d seconds.\n", this->seconds);

    printf("Per-second rate:\n");
    normalize(@func, this->seconds);
    printa(@func);

    printf("\nRaw counts:\n");
    denormalize(@func);
    printa(@func);
}
```

Wenn das obige Skript kurz ausgeführt wird, erhalten wir eine ähnliche Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./denorm.d
^C
Ran for 14 seconds.
Per-second rate:

  syslogd                                                   0
  in.routed                                                 0
  xbiff                                                     1
  sendmail                                                  2
  elm                                                       2
  picld                                                     3
  httpd                                                     4
  xclock                                                    6
  FvwmAuto                                                  7
  mibiisa                                                  22
  dtrace                                                   42
  java                                                     55
  xterm                                                    75
  adeptedit                                               118
  nscd                                                    127
  prstat                                                  179
  perfbar                                                 184
  fvwm2                                                   296
  Xsun                                                    829

Raw counts:
```

```
  syslogd                                                   1
  in.routed                                                 4
  xbiff                                                    21
  sendmail                                                 30
  elm                                                      36
  picld                                                    43
  httpd                                                    56
  xclock                                                   91
  FvwmAuto                                                104
  mibiisa                                                 314
  dtrace                                                  592
  java                                                    774
  xterm                                                  1062
  adeptedit                                              1665
  nscd                                                   1781
  prstat                                                 2506
  perfbar                                                2581
  fvwm2                                                  4156
  Xsun                                                  11616
```

Aggregate können auch renormalisiert werden. Wenn `normalize()` mehrmals für dasselbe Aggregat aufgerufen wird, gilt der im neuesten Aufruf angegebene Normalisierungsfaktor. Das folgende Beispiel gibt die Häufigkeit pro Sekunde im Verlauf der Zeit aus:

BEISPIEL 9–1 `renormalize.d`: Renormalisierung eines Aggregats

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
}

syscall:::entry
{
    @func[execname] = count();
}

tick-10sec
{
    normalize(@func, (timestamp - start) / 1000000000);
    printa(@func);
}
```

Löschen von Aggregaten

Wenn Sie mit DTrace einfache Überwachungsskripten erzeugen, können Sie die Werte in einem Aggregat mit der Funktion `clear()` regelmäßig löschen lassen. Diese Funktion nimmt als einzigen Parameter ein Aggregat an. Die Funktion `clear()` löscht nur die *Werte* des Aggregats; die Aggregatschlüssel bleiben erhalten. Ein Schlüssel mit dem Wert Null in einem Aggregat deutet folglich darauf hin, dass der Schlüssel zuvor einen Nicht-Nullwert besessen *hatte*, der bei einem `clear()`-Vorgang auf Null gesetzt wurde. Um sowohl die Werte als auch die Schlüssel eines Aggregats zu löschen, verwenden Sie die Funktion `trunc()`. Ausführliche Informationen dazu finden Sie unter „Abschneiden von Aggregaten“ auf Seite 131.

Im nächsten Beispiel fügen wir `clear()` in Beispiel 9–1 ein:

```
#pragma D option quiet

BEGIN
{
    last = timestamp;
}

syscall::::entry
{
    @func[execname] = count();
}

tick-10sec
{
    normalize(@func, (timestamp - last) / 1000000000);
    printa(@func);
    clear(@func);
    last = timestamp;
}
```

Während Beispiel 9–1 die Frequenz der Systemaufrufe über die Lebensdauer des `dtrace`-Aufrufs zeigt, gibt das obige Beispiel nur die Rate für die letzten zehn Sekunden aus.

Abschneiden von Aggregaten

Bei der Betrachtung von Aggregatergebnissen interessieren häufig nur die oberen Resultate. Die zu allen anderen außer den höchsten Werten gehörenden Schlüssel und Werte sind irrelevant. Auch kann es nützlich sein, durch Entfernen der Schlüssel *und* der Werte ein ganzes Aggregatergebnis zu löschen. Zu beiden Zwecken dient die DTrace-Funktion `trunc()`.

Die Parameter für `trunc()` sind ein Aggregat und ein optionaler Kürzungswert. Ohne Kürzungswert verwirft `trunc()` *sowohl* die Aggregatwerte *als auch* die Aggregatschlüssel für das gesamte Aggregat. Ist ein Kürzungswert *n* vorhanden, löscht `trunc()` die Aggregatwerte

und -schlüssel *mit Ausnahme* der Werte und Schlüssel, die zu den *n* höchsten Werten gehören. Das heißt, dass `trunc(@foo, 10)` das Aggregat namens `foo` nach den zehn höchsten Werten abschneidet, während `trunc(@foo)` das gesamte Aggregat löscht. Es wird auch dann das gesamte Aggregat gelöscht, wenn als Kürzungswert `0` angegeben wurde.

Um anstelle der *n* oberen die *n* unteren Werte anzuzeigen, übergeben Sie der Funktion `trunc()` einen negativen Kürzungswert. So schneidet beispielsweise `trunc(@foo, -10)` das Aggregat `foo` hinter den zehn niedrigsten Werten ab.

Erweitern wir nun das Systemaufruf-Beispiel so, dass über einen Zeitraum von zehn Sekunden die Häufigkeit der Systemaufrufe pro Sekunde für die zehn oberen aufrufenden Anwendungen angezeigt wird:

```
#pragma D option quiet

BEGIN
{
    last = timestamp;
}

syscall::entry
{
    @func[execname] = count();
}

tick-10sec
{
    trunc(@func, 10);
    normalize(@func, (timestamp - last) / 1000000000);
    printa(@func);
    clear(@func);
    last = timestamp;
}
```

Das nächste Beispiel zeigt die Ausgabe des obigen Skripts bei Ausführung auf einem Laptop mit geringer Systemlast:

```
  FvwmAuto                                                  7
  telnet                                                   13
  ping                                                     14
  dtrace                                                   27
  xclock                                                   34
  MozillaFirebird-                                         63
  xterm                                                   133
  fvwm2                                                   146
  acroread                                                168
  Xsun                                                    616
```

```
telnet                                                  4
FvwmAuto                                                5
ping                                                   14
dtrace                                                 27
xclock                                                 35
fvwm2                                                  69
xterm                                                  70
acroread                                              164
MozillaFirebird-                                      491
Xsun                                                 1287
```

Minimieren von Auslassungen

Da DTrace einige Aggregatdaten im Kernel zwischenspeichert, kann es unter Umständen vorkommen, dass nach Hinzufügen eines neuen Schlüssels für ein Aggregat nicht genügend Speicherplatz zur Verfügung steht. In diesem Fall werden die Daten ausgelassen, ein Zähler wird erhöht, und `dtrace` generiert eine Meldung über die Aggregatauslassung. Diese Situation ist eher unwahrscheinlich, da DTrace den Dauerstatus (bestehend aus Aggregatschlüssel und Zwischenergebnis) auf Benutzerebene hält, wo der Speicherplatz dynamisch erweitert werden kann. Sollten tatsächlich Aggregatauslassungen auftreten, können Sie die Puffergröße für Aggregate mit der Option `aggsize` erhöhen, um dieses Risiko herabzusetzen. Diese Option ermöglicht außerdem die Minimierung des Speicherplatzbedarfs von DTrace. Wie jede Größenoption kann auch `aggsize` mit jedem Größensuffix angegeben werden. Die Richtlinien zur Veränderung der Größe dieses Puffers wird von der Option `bufresize` vorgegeben. Weitere Informationen zur Pufferung finden Sie in Kapitel 11, „Puffer und Pufferung“. Weitere Informationen zu Optionen finden Sie in Kapitel 16, „Optionen und Tunables “.

Eine alternative Methode zur Vermeidung von Aggregatauslassungen besteht darin, die Häufigkeit zu erhöhen, mit der die Aggregatdaten auf Benutzerebene verbraucht werden. Standardmäßig beläuft sich diese Rate auf einmal pro Sekunde, sie kann aber mit der Option `aggrate` angepasst werden. Wie jede Frequenzoption kann auch `aggrate` mit einem beliebigen Zeitsuffix angegeben werden. Standardmäßig gilt Häufigkeit pro Sekunde. Weitere Informationen zur Option `aggsize` finden Sie in Kapitel 16, „Optionen und Tunables “.

◆ ◆ ◆ KAPITEL 10

10 Aktionen und Subroutinen

Mit D-Funktionsaufrufen wie `trace()` und `printf()` lassen sich zwei verschiedene Arten von DTrace-Diensten aufrufen: *Aktionen*, die Daten verfolgen oder den DTrace-externen Status ändern, und *Subroutinen*, die nur den internen DTrace-Status beeinflussen. In diesem Kapitel werden die Aktionen und Subroutinen definiert und ihre Syntax und Semantik beschrieben.

Aktionen

Aktionen machen die Interaktion zwischen Ihren DTrace-Programmen und dem System außerhalb von DTrace möglich. Die üblichsten Aktionen bestehen in der Aufzeichnung von Daten in einem DTrace-Puffer. Darüber hinaus stehen Aktionen wie das Anhalten des aktuellen Prozesses, das Setzen eines bestimmten Signals auf den laufenden Prozess oder das Beenden der gesamten Ablaufverfolgung zur Verfügung. Einige dieser Aktionen sind insofern als *destruktiv* zu bezeichnen, als sie das System ändern, wenn auch auf genau definierte Weise. Diese Aktionen können nur eingesetzt werden, wenn destruktive Aktionen ausdrücklich zugelassen wurden. Die Aufzeichnungsaktionen zeichnen die Daten standardmäßig im *Hauptpuffer* auf. Ausführliche Informationen zu Richtlinien für Hauptpuffer und Puffer finden Sie in Kapitel 11, „Puffer und Pufferung“.

Standardaktion

Eine Klausel kann eine beliebige Anzahl von Aktionen und Variablenmanipulationen enthalten. Wird eine Klausel leer gelassen, so erfolgt die *Standardaktion.* Die Standardaktion besteht in der Aufzeichnung der ID des aktivierten Prüfpunkts (EPID) im Hauptpuffer. Die EPID bezeichnet eine bestimmte Aktivierung eines bestimmten Prüfpunkts mit einem bestimmten Prädikat und den dazugehörigen Aktionen. Aus der EPID können DTrace-Verbraucher den Prüfpunkt ableiten, der eine Aktion eingeleitet hat. Immer wenn Daten verfolgt werden, müssen diese durch die EPID ergänzt werden, damit sie überhaupt einen Sinn für den Verbraucher ergeben. Deshalb besteht die Standardaktion darin, ausschließlich die EPID zu verfolgen.

Die Nutzung der Standardaktion ermöglicht eine einfache Verwendung von dtrace(1M). Beispielsweise aktiviert der folgende Befehl alle Prüfpunkte im Timeshare-Scheduling-Modul TS mit der Standardaktion:

```
# dtrace -m TS
```

Dieser Befehl kann eine Ausgabe wie in folgendem Beispiel erzeugen:

```
# dtrace -m TS
dtrace: description 'TS' matched 80 probes
CPU     ID                    FUNCTION:NAME
  0  12077                 ts_trapret:entry
  0  12078                ts_trapret:return
  0  12069                   ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12081                  ts_wakeup:entry
  0  12082                 ts_wakeup:return
  0  12069                    ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12069                    ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12033                  ts_setrun:entry
  0  12034                 ts_setrun:return
  0  12069                    ts_sleep:entry
  0  12070                  ts_sleep:return
  0  12023                  ts_update:entry
  0  12079             ts_update_list:entry
  0  12080            ts_update_list:return
  0  12079             ts_update_list:entry
...
```

Daten aufzeichnende Aktionen

Die Daten aufzeichnenden Aktionen stellen die zentralen Aktionen von DTrace dar. Sie alle zeichnen standardmäßig Daten im Hauptpuffer auf, ermöglichen aber auch das Aufzeichnen von Daten in spekulativen Puffern. Ausführliche Informationen zum Hauptpuffer finden Sie in Kapitel 11, „Puffer und Pufferung“. Ausführliche Informationen zu spekulativen Puffern finden Sie in Kapitel 13, „Spekulative Ablaufverfolgung“. Die Beschreibungen in diesem Abschnitt beziehen sich lediglich auf den *Zielpuffer*, wobei angegeben wird, ob die Daten im Hauptpuffer oder, wenn auf die Aktion die Funktion `speculate()` folgt, in einem spekulativen Puffer aufgezeichnet wird.

trace()

`void trace(`*Ausdruck*`)`

Die grundlegende Aktion ist `trace()`, die einen D-Ausdruck als Argument übernimmt und das Ergebnis im Zielpuffer aufzeichnet. Die folgenden Anweisungen sind Beispiele für `trace()`-Aktionen:

```
trace(execname);
trace(curlwpsinfo->pr_pri);
trace(timestamp / 1000);
trace(‘lbolt);
trace("somehow managed to get here");
```

tracemem()

`void tracemem(`*Adresse*`, size_t` *Anzahl-Byte*`)`

Die Aktion `tracemem()` übernimmt als erstes Argument einen D-Ausdruck, *Adresse*, und als zweites Argument eine Konstante, *Anzahl-Byte*. `tracemem()` kopiert den Speicherinhalt aus der mit *Adresse* angegebenen Adresse über die mit *Anzahl_Byte* angegebene Länge in den Zielpuffer.

printf()

`void printf(string` *Format*`, ...)`

Wie `trace()` verfolgt auch die Aktion `printf()` D-Ausdrücke. `printf()` ermöglicht jedoch eine gezielte Formatierung im `printf(3C)`-Stil. Wie bei `printf(3C)` bestehen die Parameter aus einer *Format*-Zeichenkette gefolgt von einer variablen Anzahl von Argumenten. Die Argumente werden standardmäßig im Zielpuffer aufgezeichnet. Anschließend werden die Argumente für die Ausgabe durch `dtrace(1M)` gemäß der angegebenen Format-Zeichenkette formatiert. So ließen sich etwa die ersten zwei Beispiele für `trace()` unter „`trace()`“ auf Seite 137 in einer einzigen `printf()`-Aktion kombinieren:

```
printf("execname is %s; priority is %d", execname, curlwpsinfo->pr_pri);
```

Weitere Informationen zu `printf()` finden Sie in Kapitel 12, „Formatierung der Ausgabe“.

printa()

`void printa(`*Aggregation*`)`
`void printa(string` *Format*`,` *Aggregation*`)`

Die Aktion `printa()` dient zum Anzeigen und Formatieren von Aggregaten. Weitere Informationen zu Aggregationen finden Sie in Kapitel 9, „Aggregate“. Wenn kein *Format* angegeben wird, verfolgt `printa()` lediglich eine Direktive, eine Anweisung für den DTrace-Verbraucher, die besagt, dass das angegebene Aggregat verarbeitet und im Standardformat angezeigt werden soll. Wenn ein *Format* angegeben wurde, wird das Aggregat gemäß der Angabe formatiert. Kapitel 12, „Formatierung der Ausgabe“ enthält eine ausführlichere Beschreibung der `printa()`-Formatzeichenkette.

`printa()` zeichnet nur eine *Direktive* auf, die besagt, dass das Aggregat vom DTrace-Verbraucher verarbeitet werden soll. Das Aggregat im Kernel wird von der Aktion nicht verarbeitet. Deshalb hängt die Dauer zwischen der Ablaufverfolgung der `printa()`-Direktive und der tatsächlichen Verarbeitung der Direktive von den die Pufferverarbeitung beeinflussenden Faktoren ab. Bei diesen Faktoren handelt es sich um die Aggregationsfrequenz, die Pufferungsregel und, wenn letztere auf `switching` gesetzt ist, die Frequenz der Pufferumschaltung. Ausführliche Beschreibungen dieser Faktoren finden Sie in Kapitel 9, „Aggregate“ und·Kapitel 11, „Puffer und Pufferung“.

stack()

```
void stack(int Anzahl_Frames)
void stack(void)
```

Die Aktion `stack()` zeichnet ein Kernel-Stackprotokoll im Zielpuffer auf. Dabei ist die Tiefe des Kernel-Stacks durch *Anzahl_Frames* vorgegeben. Wenn *Anzahl_Frames* nicht angegeben ist, werden so viele Stack-Frames aufgezeichnet, wie mit der Option `stackframes` angegeben wurden. Beispiel:

```
# dtrace -n uiomove:entry'{stack()}'
  CPU     ID                    FUNCTION:NAME
    0   9153                    uiomove:entry
                genunix‘fop_write+0x1b
                namefs‘nm_write+0x1d
                genunix‘fop_write+0x1b
                genunix‘write+0x1f7

    0   9153                    uiomove:entry
                genunix‘fop_read+0x1b
                genunix‘read+0x1d4

    0   9153                    uiomove:entry
                genunix‘strread+0x394
                specfs‘spec_read+0x65
                genunix‘fop_read+0x1b
                genunix‘read+0x1d4
   ...
```

Die Aktion `stack()` unterscheidet sich insofern leicht von anderen Aktionen, als sie auch als Schlüssel für Aggregate eingesetzt werden kann:

```
# dtrace -n kmem_alloc:entry'{@[stack()] = count()}'
dtrace: description 'kmem_alloc:entry' matched 1 probe
^C

              rpcmod`endpnt_get+0x47c
              rpcmod`clnt_clts_kcallit_addr+0x26f
              rpcmod`clnt_clts_kcallit+0x22
              nfs`rfscall+0x350
              nfs`rfs2call+0x60
              nfs`nfs_getattr_otw+0x9e
              nfs`nfsgetattr+0x26
              nfs`nfs_getattr+0xb8
              genunix`fop_getattr+0x18
              genunix`cstat64+0x30
              genunix`cstatat64+0x4a
              genunix`lstat64+0x1c
                1

              genunix`vfs_rlock_wait+0xc
              genunix`lookuppnvp+0x19d
              genunix`lookuppnat+0xe7
              genunix`lookupnameat+0x87
              genunix`lookupname+0x19
              genunix`chdir+0x18
                1

              rpcmod`endpnt_get+0x6b1
              rpcmod`clnt_clts_kcallit_addr+0x26f
              rpcmod`clnt_clts_kcallit+0x22
              nfs`rfscall+0x350
              nfs`rfs2call+0x60
              nfs`nfs_getattr_otw+0x9e
              nfs`nfsgetattr+0x26
              nfs`nfs_getattr+0xb8
              genunix`fop_getattr+0x18
              genunix`cstat64+0x30
              genunix`cstatat64+0x4a
              genunix`lstat64+0x1c
                1
    ...
```

ustack()

```
void ustack(int Anzahl_Frames, int strsize)
void ustack(int Anzahl_Frames)
void ustack(void)
```

Die Aktion `ustack()` zeichnet ein *Benutzer*-Stackprotokoll im Zielpuffer auf. Dabei ist die Tiefe des Benutzer-Stacks durch *Anzahl_Frames* vorgegeben. Wenn *Anzahl_Frames* nicht angegeben ist, werden so viele Stack-Frames aufgezeichnet, wie mit der Option `ustackframes` angegeben wurden. Während `ustack()` in der Lage ist, die Adresse der aufrufenden Frames zum Zeitpunkt der Prüfpunktauslösung zu ermitteln, werden die Stack-Frames erst in Symbole übersetzt, wenn die Aktion `ustack()` auf Benutzerebene vom DTrace-Verbraucher verarbeitet wurde. Wenn *Größe_Zeichenkette* angegeben und nicht Null ist, reserviert `ustack()` den angegebenen Speicherplatz für die Zeichenkette und nutzt ihn dazu, die Adresse direkt aus dem Kernel in ein Symbol zu übersetzen. Diese direkte Benutzersymbol-Übersetzung ist derzeit nur für Java Virtual Machines der Version 1.5 und höher verfügbar. Die Adress/Symbol-Übersetzung in Java kennzeichnet Benutzer-Stacks, die Java-Frames enthalten, mit der Java-Klasse und dem Methodennamen. Wenn diese Frames nicht übersetzt werden können, erscheinen diese nur als hexadezimale Adressen.

Im nächsten Beispiel wird ein Stack ohne Speicherplatz für die Zeichenkette und folglich ohne Java-Adress/Symbol-Übersetzung verfolgt:

```
# dtrace -n syscall::write:entry'/pid == $target/{ustack(50, 0);
    exit(0)}' -c "java -version"
dtrace: description 'syscall::write:entry' matched 1 probe
java version "1.5.0-beta3"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta3-b58)
Java HotSpot(TM) Client VM (build 1.5.0-beta3-b58, mixed mode)
dtrace: pid 5312 has exited
CPU     ID                    FUNCTION:NAME
  0     35                      write:entry
              libc.so.1`_write+0x15
              libjvm.so`__1cDhpiFwrite6FipkvI_I_+0xa8
              libjvm.so`JVM_Write+0x2f
              d0c5c946
              libjava.so`Java_java_io_FileOutputStream_writeBytes+0x2c
              cb007fcd
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
```

```
              cb002a7b
              cb002a7b
              cb002a7b
              cb002a7b
              cb000152
              libjvm.so`__1cJJavaCallsLcall_helper6FpnJJavaValue_
                          pnMmethodHandle_pnRJavaCallArguments_
                          pnGThread__v_+0x187
              libjvm.so`__1cCosUos_exception_wrapper6FpFpnJJavaValue_
                          pnMmethodHandle_pnRJavaCallArguments_
                          pnGThread__v2468_v_+0x14
              libjvm.so`__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle_
                          pnRJavaCallArguments_pnGThread __v_+0x28
              libjvm.so`__1cRjni_invoke_static6FpnHJNIEnv__pnJJavaValue_
                          pnI_jobject_nLJNICallType_pnK_jmethodID_pnSJNI_
                          ArgumentPusher_pnGThread__v_+0x180
              libjvm.so`jni_CallStaticVoidMethod+0x10f
              java`main+0x53d
```

Beachten Sie, dass die C- und C++-Stack-Frames aus der Java Virtual Machine symbolisch anhand von „verstümmelten“ C++-Symbolnamen und die Java-Stack-Frames nur als hexadezimale Adressen dargestellt werden. Das nächste Beispiel zeigt einen `ustack()`-Aufruf mit einer Zeichenkettengröße ungleich Null:

```
# dtrace -n syscall::write:entry'/pid == $target/{ustack(50, 500); exit(0)}'
      -c "java -version"
dtrace: description 'syscall::write:entry' matched 1 probe
java version "1.5.0-beta3"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0-beta3-b58)
Java HotSpot(TM) Client VM (build 1.5.0-beta3-b58, mixed mode)
dtrace: pid 5308 has exited
CPU     ID                    FUNCTION:NAME
  0     35                      write:entry
              libc.so.1`_write+0x15
              libjvm.so`__1cDhpiFwrite6FipkvI_I_+0xa8
              libjvm.so`JVM_Write+0x2f
              d0c5c946
              libjava.so`Java_java_io_FileOutputStream_writeBytes+0x2c
              java/io/FileOutputStream.writeBytes
              java/io/FileOutputStream.write
              java/io/BufferedOutputStream.flushBuffer
              java/io/BufferedOutputStream.flush
              java/io/PrintStream.write
              sun/nio/cs/StreamEncoder$CharsetSE.writeBytes
              sun/nio/cs/StreamEncoder$CharsetSE.implFlushBuffer
              sun/nio/cs/StreamEncoder.flushBuffer
              java/io/OutputStreamWriter.flushBuffer
              java/io/PrintStream.write
```

```
java/io/PrintStream.print
java/io/PrintStream.println
sun/misc/Version.print
sun/misc/Version.print
StubRoutines (1)
libjvm.so‘__1cJJavaCallsLcall_helper6FpnJJavaValue_
            pnMmethodHandle_pnRJavaCallArguments_pnGThread
            __v_+0x187
libjvm.so‘__1cCosUos_exception_wrapper6FpFpnJJavaValue_
            pnMmethodHandle_pnRJavaCallArguments_pnGThread
            __v2468_v_+0x14
libjvm.so‘__1cJJavaCallsEcall6FpnJJavaValue_nMmethodHandle
            _pnRJavaCallArguments_pnGThread__v_+0x28
libjvm.so‘__1cRjni_invoke_static6FpnHJNIEnv__pnJJavaValue_pnI
            _jobject_nLJNICallType_pnK_jmethodID_pnSJNI
            _ArgumentPusher_pnGThread__v_+0x180
libjvm.so‘jni_CallStaticVoidMethod+0x10f
java‘main+0x53d
8051b9a
```

Die obige Beispielausgabe veranschaulicht die symbolischen Stack-Frame-Informationen zu Java-Stack-Frames. Dass weiterhin hexadezimale Frames in der Ausgabe enthalten sind, ist darauf zurückzuführen, dass einige Funktionen statisch sind und keine Einträge in der Anwendungssymboltabelle für sie vorliegen. Diese Frames können nicht übersetzt werden.

Die `ustack()`-Symbolübersetzung für Java-fremde Frames erfolgt *nach* der Aufzeichnung der Stackdaten. Folglich wird der entsprechende Benutzerprozess möglicherweise beendet, noch bevor die Symbolübersetzung durchgeführt werden kann. Dadurch wird die Stack-Frame-Übersetzung unmöglich. Wenn der Benutzerprozess vor der Symbolübersetzung beendet wird, gibt `dtrace` eine Warnmeldung wie im nächsten Beispiel, gefolgt von den hexadezimalen Stack-Frames aus:

```
dtrace: failed to grab process 100941: no such process
              c7b834d4
              c7bca85d
              c7bca1a4
              c7bd4374
              c7bc2628
              8047efc
```

Verfahren zur Linderung dieses Problems werden in Kapitel 33, „Ablaufverfolgung von Benutzerprozessen“ beschrieben.

Da schließlich die DTrace-Befehle zur nachträglichen Fehleranalyse (Post-Mortem-Debugging) die Frame-Übersetzung nicht durchführen können, ergibt die Verwendung von `ustack()` mit der `ring`-Pufferregel stets unverarbeitete `ustack()`-Daten.

Das folgende D-Programm zeigt ein Beispiel für eine `ustack()`-Aktion ohne Angabe von *Größe_Zeichenkette*:

```
syscall::brk:entry
/execname == $$1/
{
    @[ustack(40)] = count();
}
```

Um dieses Beispiel auf den Webbrowser Netscape, `.netscape.bin`, in Solaris-Standardinstallationen anzuwenden, geben Sie folgenden Befehl ein:

```
# dtrace -s brk.d .netscape.bin
dtrace: description 'syscall::brk:entry' matched 1 probe
^C
                libc.so.1`_brk_unlocked+0xc
                88143f6
                88146cd
                .netscape.bin`unlocked_malloc+0x3e
                .netscape.bin`unlocked_calloc+0x22
                .netscape.bin`calloc+0x26
                .netscape.bin`_IMGCB_NewPixmap+0x149
                .netscape.bin`il_size+0x2f7
                .netscape.bin`il_jpeg_write+0xde
                8440c19
                .netscape.bin`il_first_write+0x16b
                8394670
                83928e5
                .netscape.bin`NET_ProcessHTTP+0xa6
                .netscape.bin`NET_ProcessNet+0x49a
                827b323
                libXt.so.4`XtAppProcessEvent+0x38f
                .netscape.bin`fe_EventLoop+0x190
                .netscape.bin`main+0x1875
                   1

                libc.so.1`_brk_unlocked+0xc
                libc.so.1`sbrk+0x29
                88143df
                88146cd
                .netscape.bin`unlocked_malloc+0x3e
                .netscape.bin`unlocked_calloc+0x22
                .netscape.bin`calloc+0x26
                .netscape.bin`_IMGCB_NewPixmap+0x149
                .netscape.bin`il_size+0x2f7
                .netscape.bin`il_jpeg_write+0xde
                8440c19
                .netscape.bin`il_first_write+0x16b
```

```
                8394670
                83928e5
                .netscape.bin`NET_ProcessHTTP+0xa6
                .netscape.bin`NET_ProcessNet+0x49a
                827b323
                libXt.so.4`XtAppProcessEvent+0x38f
                .netscape.bin`fe_EventLoop+0x190
                .netscape.bin`main+0x1875
                  1
    ...
```

jstack()

```
void jstack(int Anzahl_Frames, int Größe)
void jstack(int Anzahl_Frames)
void jstack(void)
```

`jstack()` ist ein Aliasname für `ustack()`. Dabei gilt als Anzahl der Stack-Frames der mit der Option `jstackframes` angegebene Wert und als Speicherplatz für die Zeichenkette der mit der Option `jstackstrsize` angegebene Wert. `jstacksize` nimmt standardmäßig einen Wert ungleich Null an. Das bedeutet, dass die Verwendung von `jstack()` einen Stack mit erfolgter Java-Frame-Übersetzung ergibt.

Destruktive Aktionen

Einige DTrace-Aktionen können als destruktiv bezeichnet werden, da sie den Zustand des Systems auf irgendeine, allerdings genau definierte Weise ändern. Destruktive Aktionen können erst nach expliziter Zulassung verwendet werden. Sie können destruktive Aktionen mit `dtrace`(1M) und der Option `-w` zulassen. Bei dem Versuch, destruktive Aktionen in `dtrace`(1M) zu aktivieren, ohne sie ausdrücklich zuzulassen, schlägt `dtrace` mit einer Meldung wie in folgendem Beispiel fehl:

```
dtrace: failed to enable 'syscall': destructive actions not allowed
```

Prozessdestruktive Aktionen

Einige destruktive Aktionen sind nur in Bezug auf einen bestimmten Prozess destruktiv. Diese Aktionen stehen Benutzern mit den Zugriffsrechten `dtrace_proc` oder `dtrace_user` zur Verfügung. Ausführliche Informationen zu DTrace-Sicherheitszugriffsrechten finden Sie in Kapitel 35, „Sicherheit“.

stop()

```
void stop(void)
```

Die Aktion `stop()` erzwingt das Anhalten des Prozesses, der den aktivierten Prüfpunkt auslöst, sobald er den Kernel das nächste Mal verlässt. Der Prozess wird dabei wie durch eine `proc(4)`-Aktion angehalten. Mit dem Dienstprogramm `prun(1)` können Prozesse, die durch die Aktion `stop()` angehalten wurden, wieder fortgesetzt werden. Die Aktion `stop()` ermöglicht das Anhalten eines Prozesses an einem beliebigen DTrace-Prüfpunkt. Sie dient dazu, bestimmte Programmzustände zu erfassen, die mithilfe eines einfachen Haltepunkts nur schwer zu erreichen wären, und dann einen herkömmlichen Debugger wie beispielsweise `mdb(1)` an den Prozess anzuhängen. Außerdem können Sie mit dem Dienstprogramm `gcore(1)` den Zustand eines angehaltenen Prozesses für die nachträgliche Analyse in einer Speicherabbilddatei speichern.

`raise()`

`void raise(int` *`Signal`*`)`

Die Aktion `raise()` sendet dem aktuell laufenden Prozess das angegebene Signal. Diese Aktion ist mit dem Befehl `kill(1)` zum Senden eines Signals an einen Prozess vergleichbar. Mit der Aktion `raise()` können Sie gezielt an einem genauen Punkt in der Ausführung eines Prozesses ein Signal senden.

`copyout()`

`void copyout(void *`*`Puffer`*`, uintptr_t` *`addr`*`, size_t` *`Anzahl_Byte`*`)`

Die Aktion `copyout()` kopiert *Anzahl_Byte* aus dem mit *Puffer* angegebenen Puffer an die mit *Adresse* angegebene Adresse im Adressraum des Prozesses, zu dem der aktuelle Thread gehört. Sollte die Benutzerraumadresse nicht auf eine gültige, durch Seitenfehler eingelagerte Seite im aktuellen Adressraum zutreffen, wird ein Fehler generiert.

`copyoutstr()`

`void copyoutstr(string` *`Zeichenkette`*`, uintptr_t` *`addr`*`, size_t` *`maxlen`*`)`

Die Aktion `copyoutstr()` kopiert die mit *Zeichenkette* angegebene Zeichenkette an die mit *Adresse* angegebene Adresse im Adressraum des Prozesses, zu dem der aktuelle Thread gehört. Sollte die Benutzerraumadresse nicht auf eine gültige, durch Seitenfehler eingelagerte Seite im aktuellen Adressraum zutreffen, wird ein Fehler generiert. Die Länge der Zeichenkette ist auf den mit der Option `strsize` festgelegten Wert beschränkt. Ausführliche Informationen finden Sie in Kapitel 16, „Optionen und Tunables “.

`system()`

`void system(string` *`Programm`*`, ...)`

Die Aktion `system()` bewirkt die Ausführung des mit *Programm* angegebenen Programms, als wäre es der Shell als Eingabe übergeben worden. Die Zeichenkette *Programm* kann beliebige

`printf()`/`printa()`-Formatumwandlungen enthalten. Es müssen Argumente angegeben werden, die mit den Formatumwandlungen übereinstimmen. Ausführliche Informationen zu zulässigen Formatumwandlungen finden Sie in Kapitel 12, „Formatierung der Ausgabe“.

Im folgenden Beispiel wird der Befehl `date(1)` einmal pro Sekunde ausgeführt:

```
# dtrace -wqn tick-1sec'{system("date")}'
 Tue Jul 20 11:56:26 CDT 2004
 Tue Jul 20 11:56:27 CDT 2004
 Tue Jul 20 11:56:28 CDT 2004
 Tue Jul 20 11:56:29 CDT 2004
 Tue Jul 20 11:56:30 CDT 2004
```

Das nächste Beispiel zeigt eine etwas komplexere Verwendung der Aktion. Hier werden neben `printf()`-Konvertierungen in der *Programm*-Zeichenkette herkömmliche Filtertools wie Pipes eingesetzt:

```
#pragma D option destructive
#pragma D option quiet

proc:::signal-send
/args[2] == SIGINT/
{
    printf("SIGINT sent to %s by ", args[1]->pr_fname);
    system("getent passwd %d | cut -d: -f5", uid);
}
```

Die Ausführung des obigen Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# ./whosend.d
SIGINT sent to MozillaFirebird- by Bryan Cantrill
SIGINT sent to run-mozilla.sh by Bryan Cantrill
^C
SIGINT sent to dtrace by Bryan Cantrill
```

Die Ausführung des angegebenen Befehls erfolgt *nicht* im Kontext des ausgelösten Prüfpunkts, sondern findet statt, wenn der Puffer, der die Angaben zur Aktion `system()` enthält, auf Benutzerebene verarbeitet wird. Wie und wann diese Verarbeitung erfolgt, hängt von der in Kapitel 11, „Puffer und Pufferung“ beschriebenen Pufferungsregel ab. Wenn die Standard-Pufferungsregel gilt, ist die Puffer-Verarbeitungsfrequenz durch die Option `switchrate` vorgegeben. Wenn Sie wie im nächsten Beispiel die `switchrate` ausdrücklich auf einen höheren als den Standardwert von 1 Sekunde einstellen, können Sie die der Aktion `system()` anhaftende Verzögerung beobachten:

```
#pragma D option quiet
#pragma D option destructive
#pragma D option switchrate=5sec
```

```
tick-1sec
/n++ < 5/
{
    printf("walltime  : %Y\n", walltimestamp);
    printf("date      : ");
    system("date");
    printf("\n");
}

tick-1sec
/n == 5/
{
    exit(0);
}
```

Die Ausführung des obigen Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./time.d
 walltime  : 2004 Jul 20 13:26:30
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:31
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:32
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:33
date      : Tue Jul 20 13:26:35 CDT 2004

walltime  : 2004 Jul 20 13:26:34
date      : Tue Jul 20 13:26:35 CDT 2004
```

Beachten Sie, dass die `walltime`-Werte voneinander abweichen, die `date`-Werte jedoch identisch sind. Dieses Ergebnis spiegelt die Tatsache wider, dass der Befehl `date(1)` nicht bei der Aufzeichnung der Aktion `system()`, sondern erst zur Verarbeitung des Puffers ausgeführt wurde.

Kerneldestruktive Aktionen

Einige destruktive Aktionen wirken sich auf das gesamte System aus. Diese Aktionen müssen natürlich mit äußerster Vorsicht eingesetzt werden, da sie jeden Prozess auf dem System und jedes andere implizit oder explizit von den Netzwerkdiensten des betreffenden Systems abhängende System beeinflussen.

breakpoint()

```
void breakpoint(void)
```

Die Aktion `breakpoint()` setzt einen Kernel-Haltepunkt, der bewirkt, dass das System anhält und dem Kernel-Debugger die Steuerung übergibt. Der Kernel-Debugger gibt eine Zeichenkette aus, die den DTrace-Prüfpunkt bezeichnet, der die Aktion ausgelöst hat. Nehmen wir beispielsweise Folgendes vor:

```
# dtrace -w -n clock:entry'{breakpoint()}'
dtrace: allowing destructive actions
dtrace: description 'clock:entry' matched 1 probe
```

Das kann auf einem SPARC-System unter Solaris zur Ausgabe folgender Meldung auf der Konsole führen:

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb 30002765700)
Type  'go' to resume
ok
```

Unter Solaris auf einem x86-System wird dies möglicherweise mit folgender Meldung auf der Konsole quittiert:

```
dtrace: breakpoint action at probe fbt:genunix:clock:entry (ecb d2b97060)
stopped at      int20+0xb:      ret
kmdb[0]:
```

Bei der Adresse im Anschluss an die Prüfpunktbeschreibung handelt es sich um die Adresse des aktivierenden Steuerblocks (ECB, Enabling Control Block) innerhalb von DTrace. Anhand dieser Adresse lassen sich weitere Informationen über die Prüfpunktaktivierung ermitteln, die die Haltepunktaktion eingeleitet hat.

Ein Fehler im Umgang mit der Aktion `breakpoint()` kann dazu führen, dass diese wesentlich häufiger als beabsichtigt aufgerufen wird. Dieses Verhalten kann es unter Umständen sogar unmöglich machen, den DTrace-Verbraucher zu beenden, der die Haltepunktaktionen auslöst. In diesem Fall sollten Sie die Kernel-Ganzzahlvariable `dtrace_destructive_disallow` auf 1 setzen. Diese Einstellung lässt *keinerlei* destruktive Aktionen auf dem Rechner zu. Greifen Sie auf diese Einstellung jedoch *ausschließlich* in der beschriebenen Situation zurück.

Das genaue Verfahren zum Festlegen von `dtrace_destructive_disallow` hängt dabei von dem jeweiligen Kernel-Debugger ab. Wenn Sie mit dem OpenBoot PROM auf einem SPARC-System arbeiten, verwenden Sie `w!`:

```
ok 1 dtrace_destructive_disallow w!
ok
```

Überprüfen Sie mit `w?`, ob die Variable gesetzt wurde:

```
ok dtrace_destructive_disallow w?
1
ok
```

Geben Sie dann go ein:

```
ok go
```

Für kmdb(1) auf einem x86- oder SPARC-System verwenden Sie den 4-Byte-write-Modifizierer (W) mit der /-Formatierung: x1 dcmd[

```
kmdb[0]: dtrace_destructive_disallow/W 1
dtrace_destructive_disallow:    0x0             =       0x1
kmdb[0]:
```

Fahren Sie fort mit :c:

```
kadb[0]: :c
```

Um anschließend wieder destruktive Aktionen zuzulassen, muss dtrace_destructive_disallow mithilfe von mdb(1) wieder auf 0 zurückgesetzt werden:

```
# echo "dtrace_destructive_disallow/W 0" | mdb -kw
dtrace_destructive_disallow:    0x1             =       0x0
#
```

panic()

```
void panic(void)
```

Wenn die Aktion panic() ausgelöst wird, verursacht sie einen Kernel-Absturz. Sie dient zum Erzwingen eines Systemspeicherabzugs zu einem gezielten Zeitpunkt. Diese Aktion eignet sich zur Untersuchung von Problemen in Kombination mit ring-Pufferung und Post-Mortem-Analyse. Weitere Informationen finden Sie in Kapitel 11, „Puffer und Pufferung“ und Kapitel 37, „Nachträgliche Ablaufverfolgung“. Beim Einsatz dieser Aktion wird eine Absturzmeldung mit der Angabe des Prüfpunkts angezeigt, der den Absturz verursacht hat. Beispiel:

```
panic[cpu0]/thread=30001830b80: dtrace: panic action at probe
syscall::mmap:entry (ecb 300000acfc8)

000002a10050b840 dtrace:dtrace_probe+518 (fffe, 0, 1830f88, 1830f88,
  30002fb8040, 300000acfc8)
  %l0-3: 0000000000000000 00000300030e4d80 0000030003418000 00000300018c0800
  %l4-7: 000002a10050b980 0000000000000500 0000000000000000 0000000000000502
000002a10050ba30 genunix:dtrace_systrace_syscall32+44 (0, 2000, 5,
  80000002, 3, 1898400)
  %l0-3: 00000300030de730 0000000002200008 00000000000000e0 000000000184d928
```

```
    %l4-7: 00000300030de000 0000000000000730 0000000000000073 0000000000000010

  syncing file systems... 2 done
  dumping to /dev/dsk/c0t0d0s1, offset 214827008, content: kernel
  100% done: 11837 pages dumped, compression ratio 4.66, dump
  succeeded
  rebooting...
```

`syslogd(1M)` gibt auch beim Neustart eine Meldung aus.

```
  Jun 10 16:56:31 machine1 savecore: [ID 570001 auth.error] reboot after panic:
  dtrace: panic action at probe syscall::mmap:entry (ecb 300000acfc8)
```

Im Meldungspuffer des Speicherabzugs sind auch der Prüfpunkt und der ECB enthalten, die für die Aktion `panic()` verantwortlich sind.

`chill()`

`void chill(int` *Nanosekunden*`)`

Die Aktion `chill()` bewirkt, dass DTrace für die angegebene Dauer in Nanosekunden in den Wartezustand versetzt wird. `chill()` ist hauptsächlich zur Untersuchung von Problemen geeignet, die mit zeitlichen Abläufen in Zusammenhang stehen könnten. Beispielsweise lassen sich mit dieser Aktion Fenster für Gleichzeitigkeitsbedingungen öffnen oder regelmäßig stattfindende Ereignisse in einen gemeinsamen oder in unterschiedliche Rhythmen schalten. Da Interrupts innerhalb des DTrace-Prüfpunktkontexts deaktiviert sind, führt jede Verwendung von `chill()` zu Interrupt-Latenz, Scheduling-Latenz und Dispatch-Latenz. `chill()` verursacht deshalb unter Umständen systemische Effekte und sollte keinesfalls willkürlich eingesetzt werden. Da die Systemtätigkeit von der regelmäßigen Interrupt-Behandlung abhängt, führt DTrace die Aktion `chill()` keinesfalls länger als 500 Millisekunden pro einsekündigem Intervall auf derselben CPU aus. Wenn das maximale `chill()`-Intervall überschritten wird, meldet DTrace wie im nächsten Beispiel einen unzulässigen Vorgang:

```
# dtrace -w -n syscall::open:entry'{chill(500000001)}'
dtrace: allowing destructive actions
dtrace: description 'syscall::open:entry' matched 1 probe
dtrace: 57 errors
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 1 (ID 14: syscall::open:entry): \
  illegal operation in action #1
```

Dieser Grenzwert wird selbst dann berücksichtigt, wenn die Zeit auf mehrere `chill()`-Aufrufe oder mehrere DTrace-Verbraucher eines einzigen Prüfpunkts aufgeteilt ist. Derselbe Fehler würde beispielsweise auch durch den folgenden Befehl generiert werden:

```
# dtrace -w -n syscall::open:entry'{chill(250000000); chill(250000001);}'
```

Besondere Aktionen

In diesem Abschnitt werden Aktionen beschrieben, die weder Daten aufzeichnen noch destruktiv sind.

Spekulative Aktionen

Für die spekulative Ablaufverfolgung stehen die Aktionen `speculate()`, `commit()` und `discard()` zur Verfügung. Diese Aktionen werden in Kapitel 13, „Spekulative Ablaufverfolgung“ erläutert.

`exit()`

`void exit(int` *Status*`)`

Die Aktion `exit()` beendet die Ablaufverfolgung unverzüglich und teilt dem DTrace-Verbraucher mit, dass er die Aufzeichnung abbrechen, die erforderliche abschließende Verarbeitung durchführen und `exit(3C)` mit dem angegebenen Status aufrufen soll. Da `exit()` einen Status an die Benutzerebene zurückgibt, ist dies zwar eine Daten aufzeichnende Aktion, aber im Gegensatz zu anderen Daten aufzeichnenden Aktionen kann `exit()` nicht spekulativ verfolgt werden. Unabhängig von der Pufferungsregel bewirkt `exit()`, dass der DTrace-Verbraucher beendet wird. Da `exit()` eine Daten aufzeichnende Aktion ist, *kann* sie verworfen werden.

Wenn `exit()` aufgerufen wird, werden nur die auf anderen CPUs bereits laufenden DTrace-Aktionen abgeschlossen. Auf keiner CPU erfolgen jedoch neue Aktionen. Die einzige Ausnahme zu dieser Regel ist die Abarbeitung des Prüfpunkts `END`, der aufgerufen wird, nachdem der DTrace-Verbraucher die Aktion `exit()` verarbeitet und angegeben hat, dass die Ablaufverfolgung zu beenden ist.

Subroutinen

Subroutinen unterscheiden sich dadurch von Aktionen, dass sie im Allgemeinen nur den internen DTrace-Status beeinflussen. Es gibt also keine destruktiven Subroutinen, und Subroutinen schreiben niemals Datenablaufprotokolle in Puffer. Viele der Subroutinen besitzen ein Pendant unter den Schnittstellen aus Teil 9F und 3C. Weitere Informationen zu den entsprechenden Subroutinen finden Sie unter `Intro(9F)` und `Intro(3)`.

`alloca()`

`void *alloca(size_t` *Groesse*`)`

`alloca()` reserviert im Scratch-Bereich die Menge der mit *Groesse* angegebenen Byte und gibt einen Zeiger auf den reservierten Speicherplatz zurück. Der zurückgegebene Zeiger hat immer eine 8-Byte-Ausrichtung. Die Gültigkeit des Scratch-Bereichs beschränkt sich auf die Lebensdauer einer Klausel. Nach Abschluss der Klausel wird die Zuweisung des mit `alloca()` reservierten Speicherplatzes wieder aufgehoben. Wenn nicht genügend Platz im Scratch-Bereich vorhanden ist, wird kein Speicherplatz reserviert und ein Fehler generiert.

basename()

`string basename(char *`*Zeichenkette*`)`

`basename()` ist ein D-Pendant zu `basename(1)`. Diese Subroutine erzeugt eine Zeichenkette, die aus einer Kopie der angegebenen Zeichenkette ohne ein auf `/` endendes Präfix besteht. Der zurückgegebenen Zeichenkette wird Speicherplatz im Scratch-Bereich reserviert, sodass sie nur für die Dauer der Klausel gültig ist. Wenn nicht genügend Platz im Scratch-Bereich vorhanden ist, wird `basename` nicht ausgeführt und ein Fehler generiert.

bcopy()

`void bcopy(void *`*src*`, void *`*dest*`, size_t` *Groesse*`)`

`bcopy()` kopiert so viele Byte wie mit *Groesse* angegeben aus dem Speicherbereich, auf den *Ausg* zeigt, in den mit *Ziel* angegebenen Speicherbereich. Der Ausgangsspeicher muss vollständig außerhalb des Scratch-Bereichs und der Zielspeicher vollständig darin liegen. Werden diese Voraussetzungen nicht erfüllt, erfolgt keine Kopie und es wird ein Fehler generiert.

cleanpath()

`string cleanpath(char *`*Zeichenkette*`)`

`cleanpath()` erzeugt eine Zeichenkette, die aus einer um gewisse redundante Teile gekürzten Kopie des mit *Zeichenkette* angegebenen Pfads besteht. Insbesondere werden „`/./`"-Elemente aus dem Pfad entfernt und „`/../`"-Elemente gekürzt. Bei der Kürzung der `/../`-Elemente im Pfad werden symbolische Links nicht berücksichtigt. Deshalb ist es denkbar, dass `cleanpath()` einen gültigen Pfad übernimmt und einen kürzeren, ungültigen zurückgibt.

Wenn die *Zeichenkette* beispielsweise „ `/foo/../bar`" ist und `/foo` ein symbolischer Link zu `/net/foo/export`, dann gibt `cleanpath()` die Zeichenkette „`/bar`" zurück, obwohl sich `bar` möglicherweise nur in `/net/foo` und nicht in `/` befindet. Diese Einschränkung ist darauf zurückzuführen, dass `cleanpath()` im Kontext eines ausgelösten Prüfpunkts aufgerufen wird, wo eine vollständige Auflösung symbolischer Links oder beliebiger Namen nicht möglich ist. Der zurückgegebenen Zeichenkette wird Speicherplatz im Scratch-Bereich reserviert, sodass sie nur für die Dauer der Klausel gültig ist. Wenn nicht genügend Platz im Scratch-Bereich vorhanden ist, wird `cleanpath` nicht ausgeführt und ein Fehler generiert.

copyin()

`void *copyin(uintptr_t` *Adr*`, size_t` *Groesse*`)`

`copyin()` kopiert die angegebene Größe in Byte von der angegebenen Benutzeradresse in einen Scratch-Puffer von DTrace und gibt die Adresse dieses Puffers zurück. Die Benutzeradresse wird als eine Adresse im Bereich des Prozesses interpretiert, zu dem der aktuelle Thread gehört. Der zurückgegebene Zeiger auf den Puffer hat immer eine 8-Byte-Ausrichtung. Die betreffende Adresse *muss* auf eine durch Seitenfehler eingelagerte Seite im aktuellen Prozess zutreffen. Ist dies nicht der Fall oder ist nur unzureichender Scratch-Platz verfügbar, wird `NULL` zurückgegeben und ein Fehler generiert. Informationen zu Verfahren, mit denen das Auftreten von `copyin`-Fehlern vermindert wird, finden Sie in Kapitel 33, „Ablaufverfolgung von Benutzerprozessen“.

copyinstr()

`string copyinstr(uintptr_t` *Adr*`)`

`copyinstr()` kopiert eine auf Null endende C-Zeichenkette von der angegebenen Benutzeradresse in einen Scratch-Puffer von DTrace und gibt die Adresse dieses Puffers zurück. Die Benutzeradresse wird als eine Adresse im Bereich des Prozesses interpretiert, zu dem der aktuelle Thread gehört. Die Länge der Zeichenkette ist auf den mit der Option `strsize` festgelegten Wert beschränkt. Näheres hierzu siehe Kapitel 16, „Optionen und Tunables “. Wie auch bei `copyin` *muss* die angegebene Adresse mit einer durch Seitenfehler eingelagerten Seite im aktuellen Prozess übereinstimmen. Ist dies nicht der Fall oder ist nur unzureichender Scratch-Platz verfügbar, wird `NULL` zurückgegeben und ein Fehler generiert. Informationen zu Verfahren, mit denen das Auftreten von Kapitel 33, „Ablaufverfolgung von Benutzerprozessen“-Fehlern vermindert wird, finden Sie in `Chapter 33, User Process Tracing`.

copyinto()

`void copyinto(uintptr_t` *Adr*`, size_t` *Groesse*`, void *`*Ziel*`)`

`copyinto()` kopiert die angegebene Menge Byte von der angegebenen Benutzeradresse in den mit *Ziel* festgelegten Scratch-Puffer von DTrace. Die Benutzeradresse wird als eine Adresse im Bereich des Prozesses interpretiert, zu dem der aktuelle Thread gehört. Die betreffende Adresse *muss* auf eine durch Seitenfehler eingelagerte Seite im aktuellen Prozess zutreffen. Ist dies nicht der Fall oder liegt auch nur ein Teil des Zielspeichers außerhalb des Scratch-Bereichs, erfolgt keine Kopie und ein Fehler wird generiert. Informationen zu Verfahren, mit denen das Auftreten von Kapitel 33, „Ablaufverfolgung von Benutzerprozessen“-Fehlern vermindert wird, finden Sie in `Chapter 33, User Process Tracing`.

dirname()

```
string dirname(char *Zeichenkette)
```

`dirname()` ist ein D-Pendant zu `dirname(1)`. Diese Subroutine erzeugt eine Zeichenkette, die aus dem mit *Zeichenkette* angegebenen Pfadnamen ohne dem letzten Glied besteht. Der zurückgegebenen Zeichenkette wird Speicherplatz im Scratch-Bereich reserviert, sodass sie nur für die Dauer der Klausel gültig ist. Wenn nicht genügend Platz im Scratch-Bereich vorhanden ist, wird `dirname` nicht ausgeführt und ein Fehler generiert.

msgdsize()

```
size_t msgdsize(mblk_t *Meldungszeiger)
```

`msgdsize()` gibt die Anzahl der Byte in der Datenmeldung zurück, auf die *Meldungszeiger* zeigt. Ausführliche Informationen finden Sie im Abschnitt `msgdsize(9F)`. `msgdsize()` berücksichtigt in der Zählung nur Datenblöcke des Typs `M_DATA`.

msgsize()

```
size_t msgsize(mblk_t *Meldungszeiger)
```

`msgsize()` gibt die Anzahl der Byte in der Meldung zurück, auf die *Meldungszeiger* zeigt. Anders als `msgdsize()` gibt `msgsize()` nicht nur die Anzahl der *Datenbyte*, sondern die *Gesamtbyteanzahl* der Meldung zurück.

mutex_owned()

```
int mutex_owned(kmutex_t *Mutex)
```

`mutex_owned()` ist eine Implementierung von `mutex_owned(9F)`. `mutex_owned()` gibt einen Wert ungleich Null zurück, wenn der aufrufende Thread derzeit den angegebenen Kernel-Mutex belegt, und Null, wenn der angegebene adaptive Mutex derzeit frei ist.

mutex_owner()

```
kthread_t *mutex_owner(kmutex_t *Mutex)
```

`mutex_owner()` gibt den Thread-Zeiger des aktuellen Besitzers des angegebenen adaptiven Kernel-Mutex zurück. `mutex_owner()` gibt `NULL` zurück, wenn der angegebene adaptive Mutex derzeit frei ist oder es sich bei dem angegebenen Mutex um eine Warteschleife (Spinlock) handelt. Siehe hierzu `mutex_owned(9F)`.

mutex_type_adaptive()

```
int mutex_type_adaptive(kmutex_t *Mutex)
```

`mutex_type_adaptive()` gibt einen Wert ungleich Null zurück, wenn es sich bei dem angegebenen Kernel-Mutex um den Typ `MUTEX_ADAPTIVE` handelt, anderenfalls gibt die Aktion Null zurück. Ein Mutex ist dann adaptiv, wenn er mindestens eine der folgenden Bedingungen erfüllt:

- Der Mutex ist statisch deklariert.
- Der Mutex wird mit einem Interrupt-Block-Cookie mit dem Wert NULL erzeugt.
- Der Mutex wird mit einem Interrupt-Block-Cookie erzeugt, das nicht mit einem hohen Interrupt übereinstimmt.

`mutex_init(9F)` enthält weitere Informationen zu Mutexen. Die meisten Mutexe im Solaris-Kernel sind adaptiv.

progenyof()

```
int progenyof(pid_t PID)
```

`progenyof()` gibt einen Wert ungleich Null zurück, wenn der aufrufende Prozess (der Prozess, zu dem der Thread gehört, der derzeit den übereinstimmenden Prüfpunkt auslöst) ein Nachkomme des mit der ID angegebenen Prozesses ist.

rand()

```
int rand(void)
```

`rand()` gibt eine pseudo-zufällige Ganzzahl zurück. Die zurückgegebene Zahl ist eine schwache, pseudo-zufällige Zahl, die in keiner kryptographischen Anwendung verwendet werden sollte.

rw_iswriter()

```
int rw_iswriter(krwlock_t *rwlock)
```

`rw_iswriter()` gibt einen Wert ungleich Null zurück, wenn die angegebene Leser/Schreiber-Sperre entweder von einem Schreiber belegt oder angefordert wird. Wird die Sperre nur von Lesern belegt und ist kein Schreiber blockiert oder ist die Sperre ganz frei, gibt `rw_iswriter()` Null zurück. Siehe hierzu `rw_init(9F)`.

rw_write_held()

```
int rw_write_held(krwlock_t *rwlock)
```

`rw_write_held()` gibt einen Wert ungleich Null zurück, wenn die angegebene Leser/Schreiber-Sperre derzeit im Besitz eines Schreibers ist. Ist die Sperre entweder nur von Lesern belegt oder ganz frei, gibt `rw_write_held()` Null zurück. Siehe hierzu `rw_init(9F)`.

speculation()

```
int speculation(void)
```

`speculation()` reserviert einen spekulativen Tracing-Puffer für die Verwendung durch `speculate()` und gibt eine ID für diesen Puffer zurück. Ausführliche Informationen finden Sie in Kapitel 13, „Spekulative Ablaufverfolgung“.

strjoin()

```
string strjoin(char *Zeichenkette1, char *Zeichenkette2)
```

`strjoin()` erzeugt eine Zeichenkette, die aus einer Verkettung von *Zeichenkette1* und *Zeichenkette2* besteht. Der zurückgegebenen Zeichenkette wird Speicherplatz im Scratch-Bereich reserviert, sodass sie nur für die Dauer der Klausel gültig ist. Wenn nicht genügend Platz im Scratch-Bereich vorhanden ist, wird `strjoin` nicht ausgeführt und ein Fehler generiert.

strlen()

```
size_t strlen(string Zeichenkette)
```

`strlen()` gibt die Länge der angegebenen Zeichenkette in Byte ohne das abschließende Null-Byte zurück.

◆ ◆ ◆ KAPITEL 11

11 Puffer und Pufferung

Mit der Pufferung und Verwaltung von Daten stellt das DTrace-Framework seinen Clients wie `dtrace(1M)` einen zentralen Dienst zur Verfügung. In diesem Kapitel wird die Datenpufferung ausführlich unter die Lupe genommen. Darüber hinaus werden Optionen beschrieben, die es ermöglichen, die DTrace-Richtlinien für die Pufferverwaltung von DTrace zu ändern.

Hauptpuffer

Der *Hauptpuffer* ist bei jedem Aufruf von DTrace vorhanden. In ihm zeichnen die Ablaufverfolgungsaktionen standardmäßig ihre Daten auf. Dabei handelt es sich um folgende Aktionen:

`exit()`	`printf()`	`trace()`	`ustack()`
`printa()`	`stack()`	`tracemem()`	

Die Hauptpuffer werden *immer* auf CPU-Basis (je CPU) zugewiesen. Diese Richtlinie kann nicht geändert werden. Es ist aber möglich, die Ablaufverfolgung und Pufferzuweisung mit der Option `cpu` auf eine einzige CPU zu beschränken.

Richtlinien für den Hauptpuffer

Mit DTrace lassen sich Abläufe in räumlich extrem beschränkten Kontexten im Kernel verfolgen. Insbesondere erlaubt DTrace die Überwachung auch in einem Kontext, in dem die Kernelsoftware keine zuverlässige Speicherplatzreservierung vornehmen kann. Als Folge dieser Flexibilität in Bezug auf den Kontext besteht *immer* die Möglichkeit, dass DTrace die Daten zu verfolgen versucht, wenn kein Speicherplatz zur Verfügung steht. Für den Umgang mit einer solchen Situation muss DTrace eine Richtlinie besitzen, die Sie allerdings auf die Bedürfnisse eines bestimmten Experiments abstimmen können. In manchen Fällen kann die Richtlinie

angemessen sein, neue Daten zu verwerfen. In anderen Fällen ist es unter Umständen wünschenswert, den durch die ältesten aufgezeichneten Daten belegten Speicherplatz für neue Daten zu verwenden. Meistens soll mit einer Richtlinie dafür gesorgt werden, die Wahrscheinlichkeit zu verringern, dass der verfügbare Speicherplatz überhaupt erst aufgebraucht wird. Für diese unterschiedlichen Anforderungen unterstützt DTrace mehrere Pufferrichtlinien. Die Unterstützung wird mit der Option `bufpolicy` implementiert und kann je Verbraucher festgelegt werden. Weitere Informationen zum Setzen von Optionen finden Sie in Kapitel 16, „Optionen und Tunables “.

Die Richtlinie `switch`

Standardmäßig gilt für den Hauptpuffer die Pufferrichtlinie `switch`. Unter dieser Richtlinie wird je CPU ein Pufferpaar zugewiesen: Ein Puffer ist aktiv, der andere inaktiv. Wenn ein DTrace-Verbraucher versucht, in einem Puffer zu lesen, vertauscht der Kernel zuerst den inaktiven und den aktiven Puffer (engl. *switch*). Dieser Wechsel der Puffer erfolgt so, dass kein Zeitfenster entsteht, in dem überwachte Daten verloren gehen könnten. Nach dem Wechsel der Puffer wird der jetzt inaktive Puffer an den DTrace-Verbraucher kopiert. Diese Richtlinie gewährleistet, dass der Verbraucher stets einen in sich stimmigen Puffer sieht: Es werden nie gleichzeitig Daten in einen Puffer geschrieben und aus ihm kopiert. Durch diese Technik werden außerdem Zeitfenster verhindert, in welchen die Ablaufverfolgung pausiert oder auf andere Weise behindert wird. Die Frequenz, mit der die Puffer vertauscht und ein Puffer ausgelesen wird, steuert der Verbraucher mithilfe der Option `switchrate`. Wie jede Frequenzoption kann auch `switchrate` mit einem beliebigen Zeitsuffix angegeben werden. Standardmäßig gilt Häufigkeit pro Sekunde. Weitere Informationen zu `switchrate` und anderen Optionen finden Sie in Kapitel 16, „Optionen und Tunables “.

Hinweis – Damit der Hauptpuffer auf Benutzerebene schneller als einmal pro Sekunde (der Standardwert) verarbeitet werden kann, sollten Sie den Wert von `switchrate` entsprechend anpassen. Die System verarbeitet Aktionen, die Aktivitäten auf Benutzerebene einleiten (wie z. B. `printa()` und `system()`), wenn der entsprechende Datensatz im Hauptpuffer verarbeitet wird. Der Wert von `switchrate` legt die Geschwindigkeit fest, mit der das System solche Aktionen verarbeiten kann.

Wenn unter der Richtlinie `switch` ein bestimmter aktivierter Prüfpunkt mehr Daten protokolliert, als Speicherplatz im aktiven Hauptpuffer verfügbar ist, werden die Daten *ausgelassen* und ein entsprechender Zähler für die jeweilige CPU wird erhöht. Kommt es zu einer oder mehreren solcher Auslassungen (engl. drops), zeigt `dtrace(1M)` eine Meldung wie in folgendem Beispiel an:

```
dtrace: 11 drops on CPU 0
```

Wenn eine Aufzeichnung die Gesamtgröße des Puffers übersteigt, wird diese unabhängig von der Pufferrichtlinie ausgelassen. Auslassungen können Sie reduzieren oder ganz umgehen,

indem Sie entweder mit der Option `bufsize` die Größe des Hauptpuffers oder mit der Option `switchrate` die Frequenz des Pufferwechsels erhöhen.

Im Rahmen der Richtlinie `switch` wird dem Scratch-Bereich für `copyin()`, `copyinstr()` und `alloca()` Speicherplatz außerhalb des aktiven Puffers zugewiesen.

Die Richtlinie `fill`

Für bestimmte Probleme bietet sich der Einsatz eines einzelnen Puffers im Kernel an. Dieser Ansatz, der sich mit der Richtlinie `switch` und geeigneten D-Konstrukten durch Inkrementierung einer Variable in D und den angemessenen Einsatz einer `exit()`-Aktion implementieren lässt, schaltet die Möglichkeit von Auslassungen jedoch nicht ganz aus. Um einen einzelnen, großen Puffer im Kernelinneren anzufordern und die Ablaufverfolgung fortzusetzen, bis mindestens einer der Puffer auf CPU-Ebene angefüllt ist, verwenden Sie die Pufferrichtlinie `fill`. Sie bewirkt, dass die Ablaufverfolgung fortgesetzt wird, bis ein aktivierter Prüfpunkt versucht, mehr Daten zu protokollieren, als in den verbleibenden Speicherplatz des Hauptpuffers hineinpassen. Wenn nur unzureichender Speicherplatz übrig bleibt, wird der Puffer als gefüllt markiert und der Verbraucher wird benachrichtigt, dass mindestens einer seiner Puffer auf CPU-Ebene voll ist. Sobald `dtrace(1M)` einen einzigen vollen Puffer entdeckt, wird die Ablaufverfolgung angehalten, alle Puffer werden verarbeitet und `dtrace` wird beendet. Es werden auch dann keine weiteren Daten in einem als gefüllt markierten Puffer aufgezeichnet, wenn diese eigentlich in den Puffer hineinpassen würden.

Zum Aktivieren der Richtlinie `fill` setzen Sie die Option `bufpolicy` auf `fill`. Der folgende Befehl protokolliert beispielsweise jeden Systemaufrufeintrag in einem CPU-weiten 2K-Puffer, wobei die Pufferrichtlinie auf `fill` gesetzt ist:

```
# dtrace -n syscall:::entry -b 2k -x bufpolicy=fill
```

Die Richtlinie `fill` und `END`-Prüfpunkte

`END`-Prüfpunkte werden normalerweise erst ausgelöst, wenn die Ablaufverfolgung durch den DTrace-Verbraucher ausdrücklich angehalten wird. `END`-Prüfpunkte werden garantiert nur auf einer CPU ausgelöst. Um welche CPU es sich dabei handelt, ist jedoch unbestimmt. Bei Verwendung von `fill`-Puffern wird die Ablaufverfolgung explizit angehalten, wenn mindestens einer der CPU-weiten Hauptpuffer als gefüllt markiert wurde. Wurde die Richtlinie `fill` ausgewählt, kann der `END`-Prüfpunkt unter Umständen auf einer CPU ausgelöst werden, die einen gefüllten Puffer besitzt. Um die Ablaufverfolgung mit `END` in `fill`-Puffern unterzubringen, berechnet DTrace die potenziell von `END`-Prüfpunkten belegte Speicherplatzmenge und *subtrahiert* diesen Speicherplatz von der Größe des Hauptpuffers. Ist die Nettogröße negativ, verweigert DTrace den Start und `dtrace(1M)` gibt eine entsprechende Fehlermeldung aus:

```
dtrace: END enablings exceed size of principal buffer
```

Der Reservierungsmechanismus gewährleistet, dass ein als gefüllt gekennzeichneter Puffer stets ausreichend Speicherplatz für einen `END`-Prüfpunkt bietet.

Die Richtlinie `ring`

Die DTrace-Pufferrichtlinie `ring` erweist sich bei der Ablaufverfolgung der zu einem Ausfall führenden Ereignisse als hilfreich. Wenn die Reproduktion des Ausfalls Stunden oder gar Tage dauern kann, ist es wahrscheinlich vorzuziehen, nur die neuesten Daten beizubehalten. Sobald ein Hauptpuffer voll ist, wird die Aufzeichnung beim ersten Eintrag fortgesetzt und überschreibt dadurch die ältesten Ablaufverfolgungsdaten. Der Ringpuffer wird durch Setzen der Option `bufpolicy` auf die Zeichenkette `ring` eingerichtet:

```
# dtrace -s foo.d -x bufpolicy=ring
```

`dtrace`(1M) produziert, wenn zum Erzeugen eines Ringpuffers eingesetzt, erst dann eine Ausgabe, wenn der Prozess abgeschlossen ist. Dann wird der Ringpuffer verbraucht und verarbeitet. `dtrace` verarbeitet die einzelnen Ringpuffer in der Reihenfolge der CPUs. Innerhalb des Puffers einer CPU werden die Protokolle in chronologischer Reihenfolge von alt nach jung angezeigt. Analog zur Pufferrichtlinie `switch` wird für die Aufzeichnungen aus unterschiedlichen CPUs keine bestimmte Reihenfolge beachtet. Sollte eine solche Reihenfolge erforderlich sein, empfiehlt es sich, die Variable `timestamp` als Teil der Ablaufverfolgungsanforderung ebenfalls zu protokollieren.

Das folgende Beispiel veranschaulicht die Aktivierung der Ringpufferung mit einer `#pragma option`-Direktive:

```
#pragma D option bufpolicy=ring
#pragma D option bufsize=16k

syscall::entry
/execname == $1/
{
    trace(timestamp);
}

syscall::rexit:entry
{
    exit(0);
}
```

Sonstige Puffer

Jede DTrace-Aktivierung beinhaltet Hauptpuffer. Bestimmte DTrace-Verbraucher können über die Hauptpuffer hinaus zusätzliche Datenpuffer innerhalb des Kernels besitzen: einen *Aggregatpuffer* (siehe Kapitel 9, „Aggregate“) und einen oder mehrere *spekulative Puffer* (siehe Kapitel 13, „Spekulative Ablaufverfolgung“).

Puffergrößen

Die Größe der einzelnen Puffer lässt sich pro Verbraucher anpassen. Die folgende Tabelle zeigt die verschiedenen Optionen zum Anpassen der Größe der einzelnen Puffer:

Puffer	Größenoption
Hauptpuffer	`bufsize`
Spekulative Puffer	`specsize`
Aggregatpuffer	`aggsize`

Jede Option ist auf einen Wert gesetzt, der die jeweilige Größe wiedergibt. Wie alle Größenoptionen kann dem Wert ein optionales Größensuffix angefügt werden. Ausführliche Informationen finden Sie in Kapitel 16, „Optionen und Tunables “. Um beispielsweise die Puffergröße in der Befehlszeile für `dtrace` auf 1 MB festzulegen, können Sie die Option mit `-x` setzen:

```
# dtrace -P syscall -x bufsize=1m
```

Alternativ übergeben Sie `-dtrace` die Option `b` :

```
# dtrace -P syscall -b 1m
```

Außerdem lässt sich `bufsize` auch über `#pragma D option` festlegen:

```
#pragma D option bufsize=1m
```

Die gewählte Puffergröße bestimmt die Größe des Puffers auf *jeder* CPU. Bei Verwendung der Pufferrichtlinie `switch` gibt `bufsize` die Größe *jedes* Puffers auf jeder CPU an. Die Puffergröße beträgt standardmäßig 4 MB.

Richtlinie für die Änderung der Puffergröße

Das System kann gelegentlich zu wenig freien Kernelspeicher verfügbar haben, um einem Puffer die gewünschte Menge an Speicherplatz zuzuweisen, weil entweder nicht genügend Speicher vorhanden ist oder der DTrace-Verbraucher einen der in Kapitel 16, „Optionen und Tunables “ beschriebenen abstimmbaren Grenzwerte überschritten hat. Sie können die Richtlinie für das Scheitern von Pufferspeicherzuweisungen mit der Option `bufresize` konfigurieren, die standardmäßig auf `auto` gesetzt ist. Im Rahmen der `auto`-Pufferrichtlinie für die Größenänderung wird die Größe eines Puffers so lange halbiert, bis eine erfolgreiche Speicherzuweisung stattfindet. Wenn ein Puffer nach der Speicherzuweisung kleiner ist, als angefordert wurde, generiert `dtrace(1M)` eine Meldung:

```
# dtrace -P syscall -b 4g
dtrace: description 'syscall' matched 430 probes
dtrace: buffer size lowered to 128m
...
```

oder:

```
# dtrace -P syscall'{@a[probefunc] = count()}' -x aggsize=1g
dtrace: description 'syscall' matched 430 probes
dtrace: aggregation size lowered to 128m
...
```

Alternativ können Sie nach der gescheiterten Pufferspeicherzuweisung einen manuellen Eingriff anfordern, indem Sie `bufresize` auf `manual` setzen. Bei dieser Richtlinie bedeutet eine fehlgeschlagene Speicherzuweisung, dass DTrace nicht starten kann:

```
# dtrace -P syscall -x bufsize=1g -x bufresize=manual
dtrace: description 'syscall' matched 430 probes
dtrace: could not enable tracing: Not enough space
#
```

Die Änderungsrichtlinie für die Größe *aller* Puffer, also der Haupt-, spekulativen und Aggregatpuffer, ist durch die Option `bufresize` vorgegeben.

◆ ◆ ◆ KAPITEL 12

12 Formatierung der Ausgabe

Die in DTrace integrierten Formatierungsfunktionen `printf()` und `printa()` können Sie in Ihren D-Programmen zum Formatieren der Ausgabe einsetzen. Der D-Compiler bietet Leistungsmerkmale, die in der `printf(3C)`-Bibliotheksroutine nicht enthalten sind. Deshalb empfiehlt es sich, dieses Kapitel auch dann zu lesen, wenn Sie mit `printf()` vertraut sind. Darüber hinaus behandelt dieses Kapitel das Formatverhalten der Funktion `trace()` und das Standardausgabeformat von `dtrace(1M)` für die Anzeige von Aggregaten.

`printf()`

Die Funktion `printf()` bietet zwei Fähigkeiten in Einem: Ähnlich wie die Funktion `trace()` verfolgt sie Daten, und zusätzlich kann sie die Daten und anderen Text in einem spezifischen, von Ihnen beschriebenen Format ausgeben. Die Funktion `printf()` weist DTrace an, die jedem Argument nach dem ersten Argument zugehörigen Daten zu verfolgen und die Ergebnisse gemäß den mit dem ersten `printf()`-Argument, der *Formatzeichenkette*, beschriebenen Regeln auszugeben.

Die Formatzeichenkette ist eine normale Zeichenkette (string). Sie kann beliebig viele mit dem Zeichen `%` angeführte Formatumwandlungen enthalten, die beschreiben, wie das entsprechende Argument formatiert werden soll. Die erste Konvertierung in der Formatzeichenkette bezieht sich auf das zweite Argument von `printf()`, die zweite Konvertierung auf das dritte Argument und so weiter. Text zwischen den Umwandlungen wird wörtlich wiedergegeben. Das auf das `%`-Umwandlungszeichen folgende Zeichen beschreibt das für das entsprechende Argument zu verwendende Format.

Anders als `printf(3C)` ist die DTrace-Funktion `printf()` eine integrierte Funktion, die vom D-Compiler erkannt wird. Der D-Compiler stellt mehrere nützliche Dienste für DTrace `printf()` zur Verfügung, die in der C-Bibliothek `printf()` nicht enthalten sind:

- Der D-Compiler vergleicht die Argumente mit den Umwandlungen in der Formatzeichenkette. Wenn der Typ eines Arguments mit der Formatumwandlung nicht vereinbar ist, gibt der D-Compiler eine erklärende Fehlermeldung aus.

- Der D-Compiler erfordert für `printf()`-Formatumwandlungen keine Größenpräfixe. Bei der C-Routine `printf()` muss die Größe der Argumente durch Hinzufügen von Präfixen wie beispielsweise `%ld` für `long` oder `%lld` für `long long` angegeben werden. Der D-Compiler kennt Größe und Typ der Argumente, sodass in `printf()`-Anweisungen in D auf diese Präfixe verzichtet werden kann.
- DTrace bietet zusätzliche Formatzeichen, die sowohl die Fehleranalyse als auch die Beobachtbarkeit begünstigen. So lässt sich etwa ein Zeiger mithilfe der `%a`-Formatumwandlung als Symbolname mit Versatz (Offset) darstellen.

Zum Implementieren dieser Leistungsmerkmale muss die Formatzeichenkette in der DTrace-Funktion `printf()` im D-Programm als konstante Zeichenkette angegeben sein. Formatzeichenketten dürfen keine dynamischen Variablen des Typs `string` sein.

Umwandlungsangaben

Jede Umwandlungsangabe in der Formatzeichenkette wird durch das Zeichen `%` eingeleitet, auf das der Reihe nach diese Informationen folgen:

- Keine oder mehrere *Flags* (in beliebiger Reihenfolge), die die Bedeutung der Umwandlungsangabe modifizieren (Beschreibung siehe nächster Abschnitt).
- Eine optionale Mindest*feldbreite*. Wenn der konvertierte Wert weniger Byte aufweist als die Feldbreite, wird er standardmäßig auf der linken Seite oder, bei Angabe des Flags zur Linksausrichtung (`-`) auf der rechten Seite, mit Zwischenraum aufgefüllt. Die Feldbreite kann auch mit einem Asterisk (`*`) angegeben werden. In diesem Fall wird die Feldbreite dynamisch, auf Grundlage des Werts eines zusätzlichen `int`-Arguments festgelegt.
- Eine optionale *Genauigkeit*, die angibt, wie viele Stellen bei `d`-, `i`-, `o`-, `u`-, `x`- und `X`-Umwandlungen mindestens verwendet werden müssen (das Feld wird durch vorangestellte Nullen angefüllt); die Anzahl der Stellen nach dem Grundzahlzeichen bei `e`-, `E`- und `f`-Umwandlungen, die maximale Anzahl signifikanter Stellen bei `g`- und `G`-Umwandlungen; oder die maximale Anzahl der Byte, die von der `s`-Umwandlung aus einer Zeichenkette ausgegeben werden sollen. Die Genauigkeit wird durch einen Punkt (`.`) gefolgt von einem Asterisk (`*`) (siehe unten) oder einer Zeichenkette von Dezimalzahlen angegeben.
- Eine optionale Folge von *Größenpräfixen*, die die Größe des zugehörigen Arguments angibt (Beschreibung siehe „Größenpräfixe“ auf Seite 166). Größenpräfixe sind in D nicht erforderlich. Sie werden aus Gründen der Kompatibilität mit der C-Funktion `printf()` zur Verfügung gestellt.
- Eine *Umwandlungskennung*, die den Typ der auf das Argument anzuwendenden Umwandlung angibt.

Die Funktion `printf(3C)` unterstützt auch Umwandlungsangaben der Form `%` *n* `$`, wobei *n* eine ganze Dezimalzahl ist. DTrace `printf()` unterstützt diese Art von Umwandlungsangabe nicht.

Flags

Die Umwandlungs-Flags für `printf()` werden durch Angabe in beliebiger Reihenfolge von mindestens einem der folgenden Zeichen aktiviert:

`'` Der ganzzahlige Teil des Ergebnisses einer Dezimalumwandlung (`%i`, `%d`, `%u`, `%f`, `%g` oder `%G`) wird mit Tausenderzeichen formatiert; dabei handelt es sich um das nicht-monetäre Gruppierungszeichen. Einige Sprachumgebungen, einschließlich der POSIX-C-Sprachumgebung, bieten keine nicht-monetären Gruppierungszeichen zur Verwendung mit diesem Flag.

`-` Das Ergebnis der Umwandlung wird im Feld links ausgerichtet. Ohne Angabe dieses Flags wird das Umwandlungsergebnis rechts ausgerichtet.

`+` Das Ergebnis einer vorzeichenbehafteten Umwandlung beginnt immer mit einem Vorzeichen (`+` oder `-`). Wird das Flag nicht angegeben, beginnt die Umwandlung nur dann mit einem Vorzeichen, wenn ein negativer Wert konvertiert wird.

`space` Wenn das erste Zeichen einer vorzeichenbehafteten Umwandlung kein Vorzeichen ist oder eine Umwandlung mit Vorzeichen keine Zeichen ergibt, wird vor das Ergebnis ein Leerzeichen gesetzt. Treten sowohl `space`- als auch `+`-Flags auf, wird das Leerzeichen-Flag ignoriert.

`#` Der Wert wird in eine alternative Form umgewandelt, sofern für die ausgewählte Umwandlung eine Alternativform definiert ist. Die alternativen Formate für Umwandlungen werden gemeinsam mit der entsprechenden Umwandlung beschrieben.

`0` Bei `d`-, `i`-, `o`-, `u`-, `x`-, `X`-, `e`-, `E`-, `f`-, `g`- und `G`-Umwandlungen werden zur Anpassung an die Feldbreite führende Nullen (im Anschluss an etwaige Angaben von Vorzeichen oder Basis) eingefügt. Es erfolgt kein Auffüllen mit Leerzeichen. Treten sowohl `0`- als auch `-`-Flags auf, wird das `0`-Flag ignoriert. Bei Angabe einer Genauigkeit in `d`-, `i`-, `o`-, `u`-, `x`- und `X`-Umwandlungen wird das `0`-Flag ignoriert. Treten sowohl das `0`- als auch das `'`-Flag auf, werden vor dem Auffüllen mit Nullen die Gruppierungszeichen eingefügt.

Kennungen für Breite und Genauigkeit

Die minimale Feldbreite kann als Zeichenkette von Dezimalzahlen im Anschluss an ein beliebiges Flag angegeben werden. In diesem Fall wird die Feldbreite auf die angegebene Spaltenanzahl gesetzt. Die Feldbreite kann auch mit einem Asterisk (`*`) angegeben werden. Dann wird zur Ermittlung der Feldbreite auf ein zusätzliches Argument des Typs `int` zugegriffen. Um beispielsweise eine Ganzzahl `x` auf einer durch den Wert der `int`-Variable `w` bestimmten Feldbreite darzustellen, schreiben Sie die D-Anweisung:

```
printf("%*d", w, x);
```

Die Feldbreite kann auch mit dem Zeichen ? angegeben werden, das bedeutet, dass die Feldbreite auf Grundlage der zur Formatierung einer hexadezimalen Adresse im Datenmodell des Betriebssystemkerns erforderlichen Anzahl von Zeichen festzulegen ist. Beruht der Kernel auf dem 32-Bit-Datenmodell, wird die Breite auf 8, beim 64-Bit-Datenmodell auf 16 gesetzt.

Die Genauigkeit der Umwandlung kann als Zeichenkette von Dezimalzahlen im Anschluss an einen Punkt (`.`) oder durch einen Asterisk (`*`) im Anschluss an einen Punkt angegeben werden. Wird die Genauigkeit mit einem Asterisk angegeben, wird zur Ermittlung der Genauigkeit auf ein zusätzliches Argument des Typs `int` vor dem Umwandlungsargument zugegriffen. Wenn sowohl die Breite als auch die Genauigkeit mit einem Asterisk angegeben werden, sollten die `printf()`-Argumente für die Umwandlung in folgender Reihenfolge auftreten: Breite, Genauigkeit, Wert.

Größenpräfixe

Größenvorzeichen sind in ANSI-C-Programmen, die `printf(3C)` enthalten, zur Angabe von Größe und Typ des Umwandlungsarguments erforderlich. Der D-Compiler erledigt diese Verarbeitung für `printf()`-Aufrufe automatisch. Größenpräfixe werden hier also nicht benötigt. Sie stehen zwar aus Gründen der Kompatibilität mit C zur Verfügung, doch wird von ihrer Verwendung in D-Programmen ausdrücklich abgeraten, da sie den Code bei Einsatz abgeleiteter Typen an ein bestimmtes Datenmodell binden. Wenn beispielsweise eine `typedef` auf andere ganzzahlige Grundtypen in Abhängigkeit vom Datenmodell umdefiniert wird, kann keine einzelne C-Umwandlung verwendet werden, die für beide Datenmodelle funktioniert, ohne dass die beiden zugrunde liegenden Typen ausdrücklich angegeben und ein Cast-Ausdruck eingefügt oder mehrere Formatzeichenketten definiert werden. Der D-Compiler löst dieses Problem automatisch, indem er auf Größenpräfixe verzichtet und die Argumentgröße automatisch ermittelt.

Die Größenpräfixe können direkt vor den Namen der Formatumwandlung und hinter etwaige Flags, Breiten- und Genauigkeitszeichen gesetzt werden. Die Größenpräfixe lauten:

- Ein optionales `h` gibt an, dass eine nachfolgende `d`-, `i`-, `o`-, `u`-, `x`- oder `X`-Umwandlung auf den Typ `short` oder `unsigned short` angewendet wird.
- Ein optionales `l` gibt an, dass eine nachfolgende `d`-, `i`-, `o`-, `u`-, `x`- oder `X`-Umwandlung auf den Typ `long` oder `unsigned long` angewendet wird.
- Ein optionales `ll` gibt an, dass eine nachfolgende `d`-, `i`-, `o`-, `u`-, `x`- oder `X`-Umwandlung auf den Typ `long long` oder `unsigned long long` angewendet wird.
- Ein optionales `L` gibt an, dass eine nachfolgende `e`-, `E`-, `f`-, `g`- oder `G`-Umwandlung auf den Typ `long double` angewendet wird.

- Ein optionales `l` gibt an, dass eine nachfolgende `c`-Umwandlung auf ein `wint_t`-Argument und ein nachfolgendes `s`-Umwandlungszeichen auf einen Zeiger auf ein `wchar_t`-Argument angewendet wird.

Umwandlungsformate

Mit jeder Folge von Umwandlungszeichen werden null oder mehr Argumente abgerufen. Wenn der Format-Zeichenkette nicht genügend Argumente geliefert werden oder am Ende der Format-Zeichenkette weitere Argumente übrig bleiben, gibt der D-Compiler eine entsprechende Fehlermeldung aus. Bei Angabe eines undefinierten Umwandlungsformats gibt der D-Compiler ebenfalls eine Fehlermeldung aus. Die Umwandlungszeichenfolgen lauten:

`a` Der Zeiger oder das `uintptr_t`-Argument wird als Kernelsymbolname in der Form *Modul*`‘`*Symbolname* plus einem optionalen hexadezimalen Byte-Abstand ausgegeben. Sollte der Wert nicht in den durch ein bekanntes Kernelsymbol definierten Bereich fallen, wird er als hexadezimale Ganzzahl ausgegeben.

`c` Das Argument des Typs `char`, `short` oder `int` wird als ASCII-Zeichen ausgegeben.

`C` Das Argument des Typs `char`, `short` oder `int` wird als ASCII-Zeichen ausgegeben, sofern es sich um ein druckbares ASCII-Zeichen handelt. Ist dies nicht der Fall, wird das Zeichen mit der entsprechenden Ersatzdarstellung (siehe Tabelle 2–5) ausgegeben.

`d` Das Argument des Typs `char`, `short`, `int`, `long` oder `long long` wird als dezimale Ganzzahl (Grundzahl 10) ausgegeben. Wenn das Argument `signed` ist, wird es als Wert mit Vorzeichen ausgegeben. Wenn das Argument `unsigned` ist, wird es als vorzeichenloser Wert ausgegeben. Diese Umwandlung ist gleichbedeutend mit `i`.

`e, E` Das Argument des Typs `float`, `double` oder `long double` wird in den Stil `[-]`*d*`.`*ddd*`e±`*dd* umgewandelt, wobei vor dem Dezimalzeichen eine Stelle steht und die Anzahl der Stellen dahinter die Genauigkeit angibt. Das Dezimalzeichen ist nicht Null, wenn das Argument nicht Null ist. Wenn kein Genauigkeitswert angegeben wird, beträgt er standardmäßig 6. Wenn die Genauigkeit 0 beträgt und das `#`-Flag nicht angegeben wurde, wird kein Dezimalzeichen angezeigt. Das Umwandlungsformat `E` ergibt eine Zahl, in der die Hochzahl nicht mit `e`, sondern mit `E` eingeführt wird. Die Hochzahl enthält stets mindestens zwei Stellen. Der Wert wird auf die geeignete Stellenanzahl aufgerundet.

`f` Das Argument des Typs `float`, `double` oder `long double` wird in den Stil `[-]`*ddd*`.` *ddd* umgewandelt, wobei die Anzahl der Stellen hinter dem Dezimalzeichen die Genauigkeit angibt. Wenn kein Genauigkeitswert angegeben wird, beträgt er standardmäßig 6. Wenn die Genauigkeit 0 beträgt und das `#`-Flag nicht angegeben

wurde, wird kein Dezimalzeichen angezeigt. Wenn ein Dezimalzeichen erscheint, steht davor mindestens eine Stelle. Der Wert wird auf die geeignete Stellenanzahl aufgerundet.

`g, G` Das Argument des Typs `float`, `double` oder `long double` wird im Stil `f` oder `e` (oder im Stil `E` bei Verwendung des Umwandlungszeichens `G`) ausgegeben, wobei die Genauigkeit die Anzahl der signifikanten Stellen angibt. Eine explizite Genauigkeit von 0 wird als 1 interpretiert. Der jeweilige Stil hängt von dem Wert ab, der umgewandelt wird: Stil `e` (oder `E`) wird nur verwendet, wenn der aus der Umwandlung resultierende Exponent kleiner als -4 oder nicht kleiner als die Genauigkeit ist. Nullen am Schluss werden aus dem Bruchteil des Ergebnisses entfernt. Ein Dezimalzeichen erscheint nur, wenn darauf eine Stelle folgt. Wenn das Flag # angegeben wurde, werden Nullen am Schluss nicht aus dem Ergebnis entfernt.

`i` Das Argument des Typs `char`, `short`, `int`, `long` oder `long long` wird als dezimale Ganzzahl (Grundzahl 10) ausgegeben. Wenn das Argument `signed` ist, wird es als Wert mit Vorzeichen ausgegeben. Wenn das Argument `unsigned` ist, wird es als vorzeichenloser Wert ausgegeben. Diese Umwandlung ist gleichbedeutend mit `d`.

`o` Das Argument des Typs `char`, `short`, `int`, `long` oder `long long` wird als vorzeichenlose Oktalzahl (Grundzahl 8) ausgegeben. Für diese Umwandlung können sowohl `signed` als auch `unsigned` Argumente benutzt werden. Wenn das Flag # angegeben wurde, wird die Genauigkeit des Ergebnisses bei Bedarf erhöht, sodass die erste Stelle des Ergebnisses eine Null ist.

`p` Das Zeiger- oder `uintptr_t`-Argument wird als Hexadezimalzahl (Grundzahl 16) ausgegeben. D akzeptiert Zeigerargumente jeden Typs. Wenn das Flag # angegeben wurde, wird einem Ergebnis, das nicht Null ist, `0x` vorangestellt.

`s` Das Argument muss ein Vektor von `char` oder ein `string` sein. Die Byte aus dem Vektor oder `string` werden bis zu einem schließenden Nullzeichen oder bis zum Ende der Daten gelesen und als ASCII-Zeichen interpretiert und ausgegeben. Wenn keine Genauigkeit angegeben ist, wird sie als unendlich angenommen und alle Zeichen bis zu dem ersten Nullzeichen werden ausgegeben. Wenn eine Genauigkeit angegeben wurde, wird nur der Teil des Zeichenvektors ausgegeben, der in der entsprechenden Menge von Bildschirmspalten sichtbar ist. Wenn ein Argument des Typs `char *` formatiert werden muss, sollte es explizit in den Typ `string` umgewandelt werden oder den D-Operator `stringof` als Präfix erhalten, der DTrace anweist, die Byte aus der Zeichenkette zu verfolgen und zu formatieren.

`S` Das Argument muss ein Vektor von `char` oder ein `string` sein. Das Argument wird wie bei der `%s`-Umwandlung verarbeitet, aber etwaige nicht druckbare ASCII-Zeichen werden durch die entsprechende Ersatzdarstellung aus Tabelle 2–5 ersetzt.

`u` Das Argument des Typs `char`, `short`, `int`, `long` oder `long long` wird als vorzeichenlose Dezimalzahl (Grundzahl 10) ausgegeben. Für diese Umwandlung

können sowohl `signed` als auch `unsigned` Argumente benutzt werden, und das Ergebnis wird stets als `unsigned` formatiert.

`wc` Das Argument des Typs `int` wird in ein Sonderzeichen des Typs `wchar_t` (erweitertes Zeichen) umgewandelt und ausgegeben.

`ws` Das Argument muss ein Vektor von `wchar_t` sein. Die Byte aus dem Vektor werden bis zu einem schließenden Nullzeichen oder bis zum Ende der Daten gelesen und als erweiterte Zeichen interpretiert und ausgegeben. Wenn keine Genauigkeit angegeben ist, wird sie als unendlich angenommen und alle erweiterten Zeichen bis zu dem ersten Nullzeichen werden ausgegeben. Wenn eine Genauigkeit angegeben wurde, wird nur der Teil des Vektors von erweiterten Zeichen ausgegeben, der in der entsprechenden Menge von Bildschirmspalten sichtbar ist.

`x, X` Das Argument des Typs `char`, `short`, `int`, `long` oder `long long` wird als vorzeichenlose Hexadezimalzahl (Grundzahl 16) ausgegeben. Für diese Umwandlung können sowohl `signed` als auch `unsigned` Argumente benutzt werden. Bei der Umwandlungsform `x` werden die Buchstabenstellen `abcdef` verwendet. Bei der Umwandlungsform `X` werden die Buchstabenstellen `ABCDEF` verwendet. Wenn das Flag `#` angegeben wurde, wird einem Ergebnis, das nicht Null ist, `0x` (bei `%x`) oder `0X` (bei `%X`) vorangestellt.

`Y` Das Argument des Typs `uint64_t`wird als Anzahl der seit 00:00 Uhr UCT am 1. Januar 1970 verstrichenen Nanosekunden interpretiert und in der folgenden `cftime(3C)`-Form ausgegeben: "`%Y %a %b %e %T %Z`." Die aktuelle Anzahl der seit 00:00 Uhr UTC am 1. Januar 1970 abgelaufenen Nanosekunden steht in der Variable `walltimestamp` zur Verfügung.

`%` Es wird das Zeichen `%` ausgegeben. Kein Argument wird umgewandelt. Die vollständige Umwandlungsangabe muss `%%` lauten.

printa()

Die Funktion `printa()` dient zum Formatieren der Ergebnisse von Aggregaten in D-Programmen. Die Funktion wird in einer von zwei Formen aufgerufen:

```
printa(@Aggregationsname);
printa(Formatzeichenkette, @Aggregationsname);
```

Bei Verwendung der ersten Form erstellt der Befehl `dtrace(1M)` eine vollständige Momentaufnahme der Aggregatdaten und wendet auf die Ausgabe das entsprechende Standardausgabeformat für Aggregate (siehe Beschreibung in Kapitel 9, „Aggregate“) an.

Bei Verwendung der zweiten Form erstellt der Befehl dtrace(1M) ebenfalls eine vollständige Momentaufnahme der Aggregatdaten und generiert eine Ausgabe gemäß den in der *Format-Zeichenkette* angegebenen Umwandlungen. Dabei gelten die folgenden Regeln:

- Die Formatumwandlungen müssen mit der Tupelsignatur übereinstimmen, die zur Aggregaterzeugung verwendet wurde. Jedes Tupelelement darf nur einmal auftreten. Wenn wir beispielsweise eine Zählung mit den folgenden D-Anweisungen aggregieren:

  ```
  @a["hello", 123] = count();
  @a["goodbye", 456] = count();
  ```

  und dann einer Prüfpunktklausel die D-Anweisung `printa(`*Format-Zeichenkette*`, @a)` hinzufügen, erstellt `dtrace` eine Momentaufnahme der Aggregatdaten und erzeugt die Ausgabe so, als hätten wir die folgenden Anweisungen eingegeben:

  ```
  printf(Format-Zeichenkette, "hello", 123);
  printf(Format-Zeichenkette, "goodbye", 456);
  ```

  und so fort mit jedem im Aggregat definierten Tupel.

- Im Gegensatz zu `printf()` muss die Format-Zeichenkette für `printa()` nicht alle Tupelelemente enthalten. Das heißt, es ist durchaus möglich, bei einem Tupel mit drei Elementen nur eine Formatumwandlung zu verwenden. Sie können deshalb beliebige Tupelschlüssel aus der `printa()`-Ausgabe ausschließen, indem Sie diese in der Aggregatdeklaration an das Ende des Tupels verschieben und in der `printa()`-Format-Zeichenkette keine Umwandlungskennungen für sie angeben.
- Das Aggregatergebnis kann durch das zusätzliche Format-Flag `@` in die Ausgabe aufgenommen werden. Dieses ist nur im Zusammenhang mit `printa()` gültig. Das Flag `@` kann mit jeder geeigneten Formatumwandlungskennung kombiniert werden und darf in einer Format-Zeichenkette mehrmals auftreten. Das bedeutet, dass das Tupelergebnis an jeder Stelle und mehrmals in der Ausgabe auftreten kann. Welche Umwandlungskennungen für die einzelnen Aggregatfunktionen zulässig sind, ist durch den Ergebnistyp der jeweiligen Aggregatfunktion bedingt. Diese lauten:

`avg()`	`uint64_t`
`count()`	`uint64_t`
`lquantize()`	`int64_t`
`max()`	`uint64_t`
`min()`	`uint64_t`
`quantize()`	`int64_t`
`sum()`	`uint64_t`

Um beispielsweise das Ergebnis von avg() zu formatieren, können wir die Formatumwandlungen %d, %i, %o, %u oder %x anwenden. Die Funktionen quantize() und lquantize() formatieren ihre Ergebnisse nicht als Einzelwerte, sondern als ASCII-Tabellen.

Das folgende D-Programm stellt ein vollständiges Beispiel für printa() dar. Darin wird der Wert von caller mit dem Provider profile geprüft und die Ergebnisse werden anschließend in Form einer einfachen Tabelle ausgegeben:

```
profile:::profile-997
{
    @a[caller] = count();
}

END
{
    printa("%@8u %a\n", @a);
}
```

Wenn Sie dieses Programm mit dtrace ausführen, warten Sie einige Sekunden lang und drücken Sie dann Strg-C. Sie erhalten eine Ausgabe der Art:

```
# dtrace -s printa.d
^C
CPU     ID                    FUNCTION:NAME
  1      2                             :END        1 0x1
       1 ohci`ohci_handle_root_hub_status_change+0x148
       1 specfs`spec_write+0xe0
       1 0xff14f950
       1 genunix`cyclic_softint+0x588
       1 0xfef2280c
       1 genunix`getf+0xdc
       1 ufs`ufs_icheck+0x50
       1 genunix`infpollinfo+0x80
       1 genunix`kmem_log_enter+0x1e8
       ...
```

Standardformat von trace()

Wenn Sie Daten nicht mit printf(), sondern mit der Funktion trace() erfassen, gibt der Befehl dtrace die Ergebnisse in einem Standardausgabeformat aus. Sind die Daten 1, 2, 4 oder 8 Byte groß, wird das Ergebnis als ganzzahliger Dezimalwert formatiert. Bei Daten jeder anderen Größe, die eine Folge druckbarer Zeichen sind und als Byte-Folge interpretiert werden, wird das Ergebnis als ASCII-Zeichenkette ausgegeben. Bei Daten einer anderen Größe, die aber keine Folge druckbarer Zeichen sind, wird das Ergebnis als eine Folge von Byte-Werten im Hexadezimalformat ausgegeben.

◆ ◆ ◆ KAPITEL 13

Spekulative Ablaufverfolgung

Dieses Kapitel befasst sich mit der DTrace-Einrichtung für die *spekulative Ablaufverfolgung*, der Fähigkeit, Daten „verdachtsweise“ zu verfolgen und später zu entscheiden, ob sie an einen Tracing-Puffer *übergeben* oder aber *verworfen* werden. Der wichtigste Mechanismus zum Herausfiltern uninteressanter Ereignisse in DTrace ist der inKapitel 4, „D-Programmstruktur“ *besprochene* Prädikatmechanismus. Prädikate sind dann nützlich, wenn wir bereits zum Zeitpunkt einer Prüfpunktauslösung wissen, ob das Prüfpunktereignis von Interesse ist. Wenn uns beispielsweise nur eine zu einem bestimmten Prozess oder Dateibezeichner gehörende Tätigkeit interessiert, wissen wir bei der Prüfpunktauslösung, ob sich diese auf den relevanten Prozess bzw. Dateibezeichner bezieht. In anderen Situationen wissen wir aber unter Umständen erst einen Moment *nach* der Prüfpunktauslösung, ob ein bestimmtes Ereignis von Interesse ist.

Wenn beispielsweise ein Systemaufruf gelegentlich mit einem üblichen Fehlercode (z. B. `EIO` oder `EINVAL`) fehlschlägt, kann es interessant sein, den Codepfad zu untersuchen, der zu dieser Fehlerbedingung führt. Zum Erfassen des Codepfads könnten Sie jeden Prüfpunkt aktivieren - allerdings nur, wenn sich der scheiternde Aufruf so isolieren lässt, dass ein bedeutungsvolles Prädikat konstruiert werden kann. Treten die Fehler nur sporadisch auf oder sind sie nichtdeterministisch, wären Sie gezwungen, alle *potenziell* interessanten Ereignisse zu verfolgen und die Daten später nachzubearbeiten, um die Ereignisse herauszufiltern, die nichts mit dem zum Fehler führenden Codepfad zu tun haben. In diesem Fall müssen sehr viele Ereignisse überwacht werden, selbst wenn die Anzahl der interessanten Ereignisse möglicherweise relativ gering ausfällt. Die Nachbearbeitung wird dadurch erschwert.

Die Einrichtung für die spekulative Ablaufverfolgung bietet Ihnen in solchen Situationen die Möglichkeit, Daten an einer oder mehreren Prüfpunktpositionen auf Verdacht hin zu verfolgen und anschließend zu entscheiden, ob die Daten an den Hauptpuffer an einer anderen Prüfpunktposition übergeben werden sollen. So enthalten die Ablaufverfolgungsdaten nur die interessante Ausgabe, es ist keine Nachbearbeitung erforderlich, und der DTrace-Overhead wird auf ein Minimum reduziert.

Schnittstellen für die Spekulation

In der folgenden Tabelle sind die DTrace-Spekulationsfunktionen beschrieben:

TABELLE 13–1 DTrace-Spekulationsfunktionen

Funktionsname	Argumente	Beschreibung
`speculation`	Keinen	Gibt eine ID für einen neuen spekulativen Puffer zurück
`speculate`	ID	Der übrige Teil der Klausel wird in dem mit ID angegebenen spekulativen Puffer aufgezeichnet
`commit`	ID	Übergibt den mit ID angegebenen spekulativen Puffer
`discard`	ID	Verwirft den mit ID angegebenen spekulativen Puffer

Erzeugen von Spekulationen

Die Funktion `speculation()` reserviert Speicherplatz für einen spekulativen Puffer und gibt eine Spekulations-ID zurück. Die Spekulations-ID dient für die nachfolgenden Aufrufe der Funktion `speculate. ()` Spekulative Puffer sind eine endliche Ressource: Wenn zum Zeitpunkt des Aufrufs von `speculation()` kein spekulativer Puffer verfügbar ist, wird eine ID von Null zurückgegeben und der entsprechende DTrace-Fehlerzähler erhöht. Eine ID von Null ist immer ungültig, kann aber an `speculate()`, `commit()` oder `discard()` übergeben werden. Schlägt ein Aufruf von `speculation()` fehl, wird eine `dtrace`-Meldung wie in folgendem Beispiel generiert:

```
dtrace: 2 failed speculations (no speculative buffer space available)
```

Der Standardwert für die Anzahl spekulativer Puffer ist 1, kann aber bei Bedarf auf einen höheren Wert eingestellt werden. Weitere Informationen dazu finden Sie unter „Spekulationsoptionen und Abstimmung“ auf Seite 181.

Arbeiten mit Spekulationen

Wenn Sie eine Spekulation benutzen möchten, müssen Sie der Funktion `speculate()` `vor()` jeglichen Daten aufzeichnenden Aktionen in einer Klausel eine von der Funktion *speculation* zurückgegebene ID übergeben. Alle nachfolgenden Daten aufzeichnenden Aktionen in einer Klausel, die eine `speculate()`-Funktion enthält, werden spekulativ verfolgt. Sollte ein Aufruf von `speculate()` in einer D-Prüfpunktklausel hinter Daten aufzeichnenden Aktionen stehen, generiert der D-Compiler einen Kompilierungszeitfehler. Aus diesem Grund dürfen Klauseln entweder spekulative oder nicht spekulative Ablaufverfolgungsanforderungen enthalten, nicht aber beides.

Aggregataktionen, destruktive Aktionen und die Aktion `exit` sind keinesfalls spekulativ. Jeder Versuch, eine dieser Aktionen in eine Klausel aufzunehmen, die eine `speculate()`-Funktion enthält, ergibt einen Kompilierzeitfehler. `speculate()` darf nicht auf `speculate()` folgen: Pro Klausel ist nur eine Spekulation zulässig. Eine Klausel, die *nur* eine `speculate()`-Funktion enthält, nimmt eine spekulative Ablaufverfolgung der Standardaktion vor, die nach Vereinbarung nur die aktivierte Prüfpunkt-ID verfolgt. Eine Beschreibung der Standardaktion finden Sie in Kapitel 10, „Aktionen und Subroutinen“.

Im Normalfall weisen wir das Ergebnis einer `speculation()` einer thread-lokalen Variable zu und setzen diese Variable dann als nachfolgendes Prädikat für andere Prüfpunkte sowie als Argument für `speculate()` ein. Beispiel:

```
syscall::open:entry
{
    self->spec = speculation();
}

syscall:::
/self->spec/
{
    speculate(self->spec);
    printf("this is speculative");
}
```

Übergeben von Spekulationen

Spekulationen übergeben Sie mit der Funktion `commit. ()` Mit der Übergabe eines spekulativen Puffers werden dessen Daten in den Hauptpuffer kopiert. Wenn die Daten im angegebenen spekulativen Puffer den im Hauptpuffer verfügbaren Speicherplatz überschreiten, werden keine Daten kopiert und der Zähler für Auslassungen wird erhöht. Wurden in dem Puffer Daten von mehreren CPUs spekulativ aufgezeichnet, werden die spekulativen Daten auf der übergebenden CPU unverzüglich und die spekulativen Daten auf anderen CPUs erst kurz nach der `commit()`-Funktion kopiert. Folglich kann zwischen einer auf einer CPU beginnenden `commit()`-Funktion und dem Kopieren der Daten aus spekulativen Puffern in die Hauptpuffer auf allen CPUs einige Zeit vergehen. Diese Dauer überschreitet jedoch keinesfalls die durch die Bereinigungsfrequenz vorgegebene Dauer. Ausführliche Informationen dazu finden Sie unter „Spekulationsoptionen und Abstimmung“ auf Seite 181.

Ein übergebender spekulativer Puffer steht nachfolgenden `speculation()`-Aufrufen erst dann zur Verfügung, wenn jeder CPU-weite spekulative Puffer vollständig in den entsprechenden CPU-weiten Hauptpuffer kopiert wurde. In gleicher Weise werden nachfolgende Aufrufe von `speculate()` für den übergebenden Puffer kommentarlos verworfen und nachfolgende Aufrufe von `commit()` oder `discard()` schlagen kommentarlos fehl. Darüber hinaus darf eine

Klausel mit einer `commit()`-Funktion keine Daten aufzeichnenden Aktionen enthalten. Eine Klausel kann jedoch mehrere `commit()`-Aufrufe zum Übergeben nicht zusammenhängender Puffer enthalten.

Verwerfen von Spekulationen

Zum Verwerfen von Spekulationen verwenden Sie die Funktion `commit. ()` Das Verwerfen eines spekulativen Puffers bedeutet, dass sein Inhalt gelöscht wird. Wenn die Spekulation nur auf der die Funktion `discard()` aufrufenden CPU aktiv war, wird der Puffer unverzüglich für nachfolgende Aufrufe von `speculation()` verfügbar . War die Spekulation auf mehreren CPUs aktiv, wird der verworfene Puffer erst kurz nach dem `discard()`-Aufruf für nachfolgende Aufrufe von `speculation()` verfügbar. Dabei übersteigt die Zeit zwischen einem `discard()`-Aufruf auf einer CPU und der Freigabe des Puffers für nachfolgende Spekulationen garantiert nicht die durch die Bereinigungsfrequenz vorgegebene Dauer. Wenn zum Zeitpunkt des Aufrufs von `speculation()` kein Puffer verfügbar ist, da *alle* spekulativen Puffer derzeit verworfen oder übergeben werden, wird eine `dtrace`-Meldung wie in folgendem Beispiel generiert:

```
dtrace: 905 failed speculations (available buffer(s) still busy)
```

Die Wahrscheinlichkeit, dass kein Puffer verfügbar ist, lässt sich durch Anpassen der Menge der spekulativen Puffer oder der Bereinigungsfrequenz herabsetzen. Ausführliche Informationen dazu finden Sie unter „Spekulationsoptionen und Abstimmung“ auf Seite 181.

Beispiel für eine Spekulation

Spekulationen lassen sich beispielsweise zum Hervorheben eines bestimmten Codepfads einsetzen. Das folgende Beispiel zeigt den vollständigen Codepfad unter dem `open(2)()`-Systemaufruf nur dann, wenn die Funktion open fehlschlägt:

BEISPIEL 13–1 `specopen.d`: Codefluss für Fehlschlag von `open( )`

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

syscall::open:entry,
syscall::open64:entry
{
    /*
     * The call to speculation() creates a new speculation.  If this fails,
     * dtrace(1M) will generate an error message indicating the reason for
     * the failed speculation(), but subsequent speculative tracing will be
     * silently discarded.
     */
```

BEISPIEL 13–1 specopen.d: Codefluss für Fehlschlag von open( ) *(Fortsetzung)*

```
    self->spec = speculation();
    speculate(self->spec);

    /*
     * Because this printf() follows the speculate(), it is being
     * speculatively traced; it will only appear in the data buffer if the
     * speculation is subsequently commited.
     */
    printf("%s", stringof(copyinstr(arg0)));
}

fbt:::
/self->spec/
{
    /*
     * A speculate() with no other actions speculates the default action:
     * tracing the EPID.
     */
    speculate(self->spec);
}

syscall::open:return,
syscall::open64:return
/self->spec/
{
    /*
     * To balance the output with the -F option, we want to be sure that
     * every entry has a matching return.  Because we speculated the
     * open entry above, we want to also speculate the open return.
     * This is also a convenient time to trace the errno value.
     */
    speculate(self->spec);
    trace(errno);
}

syscall::open:return,
syscall::open64:return
/self->spec && errno != 0/
{
    /*
     * If errno is non-zero, we want to commit the speculation.
     */
    commit(self->spec);
    self->spec = 0;
}
```

BEISPIEL 13–1 `specopen.d`: Codefluss für Fehlschlag von `open( )` *(Fortsetzung)*

```
syscall::open:return,
syscall::open64:return
/self->spec && errno == 0/
{
    /*
     * If errno is not set, we discard the speculation.
     */
    discard(self->spec);
    self->spec = 0;
}
```

Die Ausführung des obigen Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# ./specopen.d
dtrace: script './specopen.d' matched 24282 probes
CPU FUNCTION
  1  => open                                  /var/ld/ld.config
  1    -> open
  1      -> copen
  1        -> falloc
  1          -> ufalloc
  1            -> fd_find
  1              -> mutex_owned
  1              <- mutex_owned
  1            <- fd_find
  1            -> fd_reserve
  1              -> mutex_owned
  1              <- mutex_owned
  1              -> mutex_owned
  1              <- mutex_owned
  1            <- fd_reserve
  1          <- ufalloc
  1          -> kmem_cache_alloc
  1            -> kmem_cache_alloc_debug
  1              -> verify_and_copy_pattern
  1              <- verify_and_copy_pattern
  1              -> file_cache_constructor
  1                -> mutex_init
  1                <- mutex_init
  1              <- file_cache_constructor
  1              -> tsc_gethrtime
  1              <- tsc_gethrtime
  1              -> getpcstack
  1              <- getpcstack
  1              -> kmem_log_enter
```

```
1              <- kmem_log_enter
1            <- kmem_cache_alloc_debug
1          <- kmem_cache_alloc
1          -> crhold
1          <- crhold
1        <- falloc
1        -> vn_openat
1          -> lookupnameat
1            -> copyinstr
1            <- copyinstr
1            -> lookuppnat
1              -> lookuppnvp
1                -> pn_fixslash
1                <- pn_fixslash
1                -> pn_getcomponent
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                    -> groupmember
1                      -> supgroupmember
1                      <- supgroupmember
1                    <- groupmember
1                  <- ufs_iaccess
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1                -> pn_getcomponent
1                <- pn_getcomponent
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                  <- ufs_iaccess
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1                -> pn_getcomponent
1                <- pn_getcomponent
```

```
1                -> ufs_lookup
1                  -> dnlc_lookup
1                    -> bcmp
1                    <- bcmp
1                  <- dnlc_lookup
1                  -> ufs_iaccess
1                    -> crgetuid
1                    <- crgetuid
1                  <- ufs_iaccess
1                  -> vn_rele
1                  <- vn_rele
1                <- ufs_lookup
1                -> vn_rele
1                <- vn_rele
1              <- lookuppnvp
1            <- lookuppnat
1          <- lookupnameat
1        <- vn_openat
1        -> setf
1          -> fd_reserve
1            -> mutex_owned
1            <- mutex_owned
1            -> mutex_owned
1            <- mutex_owned
1          <- fd_reserve
1          -> cv_broadcast
1          <- cv_broadcast
1        <- setf
1        -> unfalloc
1          -> mutex_owned
1          <- mutex_owned
1          -> crfree
1          <- crfree
1          -> kmem_cache_free
1            -> kmem_cache_free_debug
1              -> kmem_log_enter
1              <- kmem_log_enter
1              -> tsc_gethrtime
1              <- tsc_gethrtime
1              -> getpcstack
1              <- getpcstack
1              -> kmem_log_enter
1              <- kmem_log_enter
1              -> file_cache_destructor
1                -> mutex_destroy
1                <- mutex_destroy
1              <- file_cache_destructor
1              -> copy_pattern
```

```
1              <- copy_pattern
1            <- kmem_cache_free_debug
1          <- kmem_cache_free
1        <- unfalloc
1        -> set_errno
1        <- set_errno
1      <- copen
1    <- open
1  <= open                                          2
```

Spekulationsoptionen und Abstimmung

Sollte ein spekulativer Puffer bei einem Versuch einer spekulativen Ablaufverfolgung bereits voll sein, werden keine Daten in dem Puffer gespeichert und der Auslassungszähler wird erhöht. In diesem Fall wird eine `dtrace`-Meldung wie in folgendem Beispiel generiert:

```
dtrace: 38 speculative drops
```

Auslassungen von Spekulationen verhindern *nicht*, dass der volle spekulative Puffer in den Hauptpuffer kopiert wird, wenn eine Übergabe stattfindet. Ebenso können Ausfälle von Spekulationen selbst dann auf einem spekulativen Puffer vorkommen, wenn dieser bereits verworfen wurde. Ausfälle von Spekulationen lassen sich durch Vergrößern des spekulativen Puffers mit der Option `specsize` herabsetzen. Die Option `specsize` kann mit jedem Größensuffix angegeben werden. Die Richtlinien zur Veränderung der Größe dieses Puffers wird von der Option `bufresize` vorgegeben.

Es ist denkbar, dass beim Aufruf von `speculation()` kein spekulativer Puffer verfügbar ist. Wenn Puffer vorliegen, die noch nicht übergeben oder verworfen wurden, generiert `dtrace` eine Meldung wie in folgendem Beispiel:

```
dtrace: 1 failed speculation (no speculative buffer available)
```

Sie können die Wahrscheinlichkeit, dass Spekulationen auf diese Weise scheitern, durch Erhöhung der Anzahl spekulativer Puffer mit der Option `nspec` verringern. Der Standardwert von `nspec` beträgt 1.

Andererseits ist es auch möglich, dass `speculation()` fehlschlägt, weil alle spekulativen Puffer ausgelastet sind. In diesem Fall wird eine `dtrace`-Meldung wie in folgendem Beispiel generiert:

```
dtrace: 1 failed speculation (available buffer(s) still busy)
```

Diese Meldung weist darauf hin, dass der `speculation()`-Aufruf erfolgt ist, nachdem `commit()` für einen spekulativen Puffer aufgerufen, aber noch bevor dieser Puffer tatsächlich auf allen CPUs übergeben wurde. Sie können die Wahrscheinlichkeit, dass Spekulationen auf diese Weise scheitern, durch Erhöhung der CPU-Bereinigungsfrequenz mit der Option `cleanrate` verringern. Der Standardwert von `cleanrate` beträgt `101hz`.

Hinweis – Die Werte für die Option `cleanrate` sind in Anzahl pro Sekunde anzugeben. Verwenden Sie das Suffix `hz`.

◆ ◆ ◆ KAPITEL 14

Das Dienstprogramm dtrace(1M)

Der Befehl dtrace(1M) ist ein generisches Front-End für die DTrace-Einrichtung. Der Befehl implementiert eine einfache Schnittstelle zum Aufrufen des D-Compilers, die Fähigkeit, gepufferte Ablaufverfolgungsdaten aus der DTrace-Kerneleinrichtung abzurufen, sowie einen Satz grundlegender Routinen für die Formatierung und Ausgabe der aufgezeichneten Daten. Dieses Kapitel stellt eine vollständige Befehlsreferenz für den Befehl dtrace dar.

Beschreibung

Der Befehl dtrace fungiert als generische Schnittstelle zu allen wesentlichen Diensten der DTrace-Einrichtung. Dabei handelt es sich um:

- Optionen zum Auflisten der derzeit von DTrace veröffentlichten Prüfpunkte und Provider
- Optionen zum direkten Aktivieren von Prüfpunkten anhand beliebiger Prüfpunktbeschreibungsangaben (Provider, Modul, Funktion, Name)´
- Optionen zum Ausführen des D-Compilers und Kompilieren einer oder mehrerer D-Programmdateien oder direkt in die Befehlszeile eingegebener Programme
- Optionen zum Generieren von Programmen zur anonymen Ablaufverfolgung (siehe Kapitel 36, „Anonyme Ablaufverfolgung“)
- Optionen zum Generieren von Programmstabilitätsberichten (siehe Kapitel 39, „Stabilität“)
- Optionen zum Ändern des Überwachungs- und Pufferungsverhaltens von DTrace und Aktivieren zusätzlicher Leistungsmerkmale des D-Compilers (siehe Kapitel 16, „Optionen und Tunables “)

Außerdem bietet dtrace Ihnen die Möglichkeit, D-Skripten zu erstellen, indem Sie den Befehl in einer #!-Deklaration zum Erzeugen einer Interpreterdatei einsetzen (siehe Kapitel 15, „Scripting“). Schließlich und endlich können Sie mit dtrace versuchen, D-Programme zu kompilieren und ihre Eigenschaften zu ermitteln, ohne tatsächlich Ablaufverfolgungen zu aktivieren. Hierzu dient die weiter unten beschriebene Option -e.

Optionen

Der Befehl `dtrace` übernimmt die folgenden Optionen:

`dtrace` `[-32 | -64] [-aACeFGHlqSvVwZ] [-b` *Puffergr*`] [-c` *Befehl*`] [-D` *Name* `[=`*Def*`]] [-I` *Pfad*`] [-L` *Pfad*`] [-o` *Ausgabe*`] [-p` *PID*`] [-s` *Skript*`] [-U` *Name*`] [-x` *Arg* `[=`*Wert*`]] [-Xa | c | s | t] [-P` *Provider* `[ [`*Prädikat*`]`*Aktion*`]] [-m [ [`*Provider:*`]`*Modul* `[ [`*Prädikat*`]`*Aktion*`]]] [-f [ [`*Provider:*`]`*Modul:* `]`*Funktion* `[ [`*Prädikat*`]`*Aktion*`]] [-n [ [ [`*Provider:*`]`*Modul:*`]`*Funktion:*`]`*name* `[ [`*Prädikat*`]`*Aktion*`]] [-i` *Prüfpunkt-ID* `[ [`*Prädikat*`]`*Aktion*`]]`

wobei *Prädikat* ein beliebiges in Schrägstrichen `/ /` eingeschlossenes Prädikat und *Aktion* eine beliebige in geschweiften Klammern `{ }` eingeschlossene D-Anweisungsliste im Einklang mit der beschriebenen D-Sprachsyntax sind. Wenn den Optionen `-P`, `-m`, `-f`, `-n` oder `-i` D-Programmcode als Argument übergeben wird, muss dieser Text ordnungsgemäß·in Anführungszeichen gesetzt werden, um eine Interpretation durch die Shell zu verhindern. Die Optionen lauten:

`-32`, `-64` Der D-Compiler erzeugt Programme auf Grundlage des nativen Datenmodells des Betriebssystemkerns. Mit dem Befehl `isainfo(1)` `-b` können Sie das aktuelle Betriebssystemdatenmodell ermitteln. Bei Angabe der Option `-32` bringt `dtrace` den D-Compiler dazu, ein D-Programm auf Grundlage des 32-Bit-Datenmodells zu kompilieren. Bei Angabe der Option `-64` bringt `dtrace` den D-Compiler dazu, ein D-Programm auf Grundlage des 64-Bit-Datenmodells zu kompilieren. Diese Optionen werden in der Regel nicht benötigt, da `dtrace` standardmäßig das native Datenmodell auswählt. Das Datenmodell wirkt sich auf die Größe von Integer-Typen und andere Eigenschaften der Sprache aus. Die für eines der Datenmodelle kompilierten D-Programme können sowohl auf 32- als auch auf 64-Bit-Kernels ausgeführt werden. Die Optionen `-32` und `-64` bestimmen auch das Format der mit der Option `-G` erstellten ELF-Datei (ELF32 oder ELF64).

`-a` Anonymen Ablaufverfolgungsstatus fordern und die verfolgten Daten anzeigen. In Kombination erzwingen die Optionen `-a` und `-e` die sofortige Beendigung von `dtrace`, sobald der anonyme Ablaufverfolgungsstatus verbraucht ist, ohne auf weitere Daten zu warten. Weitere Informationen zur anonymen Ablaufverfolgung finden Sie in Kapitel 36, „Anonyme Ablaufverfolgung“.

`-A` `driver.conf(4)`-Direktiven für die anonyme Ablaufverfolgung generieren. Wenn die Option `-A` angegeben ist, kompiliert `dtrace` alle mit der Option `-s` oder in der Befehlszeile angegebenen D-Programme, erstellt einen Satz `dtrace(7D)` -Konfigurationsdateidirektiven zum Aktivieren der angegebenen Prüfpunkte für die anonyme Ablaufverfolgung (siehe Kapitel 36, „Anonyme Ablaufverfolgung“) und wird dann beendet. Standardmäßig versucht `dtrace`, die Direktiven in der Datei `/kernel/drv/dtrace.conf` zu speichern. Dieses Verhalten lässt sich durch Angabe einer alternativen Ausgabedatei mit der Option `-o` ändern.

`-b` Größe des Haupt-Ablaufverfolgungspuffers festlegen. Die Größe des Ablaufverfolgungspuffers kann ein beliebiges der Größensuffixe `k`, `m`, `g` oder `t` (siehe Beschreibung in Kapitel 36, „Anonyme Ablaufverfolgung“) enthalten. Kann der Pufferplatz nicht zugewiesen werden, versucht `dtrace`, die Puffergröße zu verringern oder wird beendet. Dies hängt von der Einstellung der Eigenschaft `bufresize` ab.

`-c` Angegebenen *Befehl* ausführen und anschließend Vorgang beenden. Wenn die Option `-c` in der Befehlszeile mehrmals vorhanden ist, wird `dtrace` nach dem Ende aller Befehle ebenfalls beendet und meldet den Beendigungsstatus für jeden untergeordneten Prozess. Die Prozess-ID des ersten Befehls wird jedem in der Befehlszeile oder mit der Option `-s` angegebene D-Programm über die Makrovariable `$target` verfügbar gemacht. Weitere Informationen zu Makrovariablen finden Sie in Kapitel 15, „Scripting“.

`-C` Vor dem Kompilieren den C-Preprozessor `cpp(1)` auf den D-Programmen ausführen. Über die Optionen `-D`, `-U`, `-I` und `-H` können dem C-Preprozessor Optionen übergeben werden. Mit der Option `-X` lässt sich der Grad an Konformität mit dem C-Standard wählen. Die vom D-Compiler beim Aufruf des C-Preprozessors definierten Symbole werden in der Beschreibung der Option `-X` erklärt.

`-D` Beim Aufruf von `cpp(1)`; (mit der Option `-C` aktiviert) den angegebenen *Name* definieren. Wenn Sie ein Gleichheitsszeichen (`=`) und einen zusätzlichen *Wert* angeben, wird dem Namen der entsprechende Wert zugewiesen. Diese Option übergibt jedem Aufruf von `cpp` die Option `-D`.

`-e` Vorgang nach dem Kompilieren etwaiger Anforderungen und nach Verbrauch des anonymen Ablaufverfolgungsstatus (Option `-a`), aber noch vor dem Aktivieren von Prüfpunkten beenden. In Kombination mit der Option `-a` dient diese Option zum Ausgeben anonymer Ablaufverfolgungsdaten und Beenden, in Kombination mit Optionen des D-Compilers ermöglicht sie das Überprüfen der Kompilierfähigkeit von Programmen, ohne diese ausführen und die entsprechende Instrumentation aktivieren zu müssen.

`-f` Namen der zu verfolgenden oder aufzulistenden (Option `-l`) Funktion angeben. Für das zugehörige Argument ist jede der Prüfpunktbeschreibungsformen *Provider:Modul:Funktion*, *Modul:Funktion* oder *Funktion* zulässig. Nicht angegebene Prüfpunktbeschreibungsfelder bleiben leer und treffen, unabhängig von ihren jeweiligen Werten in diesen Feldern, auf alle Prüfpunkte zu. Wenn in der Beschreibung keine Kennzeichner außer *Funktion* angegeben sind, trifft die Beschreibung auf alle Prüfpunkte mit der entsprechenden *Funktion* zu. Dem `-f`-Argument kann eine optionale D-Prüfpunktklausel angefügt werden. Die Option `-f` darf in der Befehlszeile mehrmals gleichzeitig verwendet werden.

`-F` Ausgabe der Ablaufverfolgung durch Angabe von Funktionseintritt und -rückkehr zusammenfassen. Meldungen des Prüfpunkts für Funktionseintritt werden eingerückt und der Ausgabe wird `->` vorangestellt. Meldungen des Prüfpunkts für Funktionsrückkehr werden nicht eingerückt und der Ausgabe wird `<-` vorangestellt.

`-G` ELF-Datei mit eingebettetem DTrace-Programm generieren. Die im Programm angegebenen DTrace-Prüfpunkte werden in einem verschiebbaren ELF-Objekt gespeichert, das in ein anderes Programm eingebunden werden kann. Wenn die Option `-o` vorhanden ist, wird die ELF-Datei unter dem als Argument für diesen Operanden angegebenen Pfadnamen gespeichert. Wenn die Option `-o` nicht vorhanden und das DTrace-Programm in einer Datei *Dateiname*`.s` enthalten ist, wird die ELF-Datei unter dem Namen *Datei*`.o` gespeichert; anderenfalls wird die ELF-Datei unter dem Namen `d.out` gespeichert.

`-H` Beim Aufruf von `cpp(1)`; (mit der Option `-C` aktiviert) die Pfadnamen der enthaltenen Dateien ausgeben. Diese Option übergibt jedem Aufruf von `-cpp` die Option `H` und bewirkt so, dass der Befehl die Liste der Pfadnamen zeilenweise an `stderr` ausgibt.

`-i` ID des zu verfolgenden oder aufzulistenden (Option (`-l`) Prüfpunkts angeben. Prüfpunkt-IDs werden, wie mit `dtrace -l` dargestellt, durch Dezimalzahlen angegeben. Dem `-i`-Argument kann eine optionale D-Prüfpunktklausel angefügt werden. Die Option `-i` darf in der Befehlszeile mehrmals gleichzeitig verwendet werden.

`-I` Beim Aufruf von `cpp(1)` (mit der Option `-C` aktiviert) den angegebenen Verzeichnis-*Pfad* in den Suchpfad für `#include`-Dateien einfügen. Diese Option übergibt jedem Aufruf von `cpp` die Option `-I.` Das angegebene Verzeichnis wird vor der Standardverzeichnisliste in den Suchpfad eingefügt.

`-l` Prüfpunkte auflisten, anstatt sie zu aktivieren. Wenn die Option `-l` angegeben ist, erzeugt `dtrace` einen Bericht der Prüfpunkte, die mit den die Optionen `-P`, `-m`, `-f`, `-n`, `-i` oder `-s` enthaltenden Beschreibungen übereinstimmen. Wenn keine dieser Optionen angegeben ist, werden alle Prüfpunkte aufgelistet.

`-L` Den angegebenen Verzeichnis*pfad* in den Suchpfad für DTrace-Bibliotheken einfügen. DTrace-Bibliotheken dienen zur Aufnahme geläufiger Definitionen, die beim Schreiben von D-Programmen verwendet werden können. Der angegebene *Pfad* wird hinter dem Standardsuchpfad für Bibliotheken eingefügt.

`-m` Namen des zu verfolgenden oder aufzulistenden (Option (`-l`) Moduls angeben. Für das zugehörige Argument ist jede der Prüfpunktbeschreibungsformen *Provider:Modul* oder *Modul* zulässig. Nicht angegebene Prüfpunktbeschreibungsfelder bleiben leer und treffen, unabhängig von ihren jeweiligen Werten in diesen Feldern, auf alle Prüfpunkte zu. Wenn in der Beschreibung keine Kennzeichner außer *Modul* angegeben sind, trifft die

Beschreibung auf alle Prüfpunkte mit dem entsprechenden *Modul* zu. Dem `-m`-Argument kann eine optionale D-Prüfpunktklausel angefügt werden. Die Option `-m` darf in der Befehlszeile mehrmals gleichzeitig verwendet werden.

`-n` Name des zu verfolgenden oder aufzulistenden (Option (`-l`) Prüfpunkts angeben. Für das zugehörige Argument ist jede der Prüfpunktbeschreibungsformen *Provider:Modul:Funktion:Name*, *Modul:Funktion:Name*, *Funktion:Name* oder *Name* zulässig. Nicht angegebene Prüfpunktbeschreibungsfelder bleiben leer und treffen, unabhängig von ihren jeweiligen Werten in diesen Feldern, auf alle Prüfpunkte zu. Wenn in der Beschreibung keine Kennzeichner außer *Name* angegeben sind, trifft die Beschreibung auf alle Prüfpunkte mit dem entsprechenden *Namen* zu. Dem `-n`-Argument kann eine optionale D-Prüfpunktklausel angefügt werden. Die Option `-n` darf in der Befehlszeile mehrmals gleichzeitig verwendet werden.

`-o` *Ausgabe*datei für die Optionen `-A`, `-G` und `-l` oder für die verfolgten Daten angeben. Wenn zwar die Option `-A`, nicht aber die Option `-o` vorhanden ist, gilt `/kernel/drv/dtrace.conf` als Standardausgabedatei. Wenn die Option `-G` vorhanden ist, das Argument der Option `-s` die Form *Dateiname*`.d` hat und `-o` nicht vorhanden ist, gilt *Dateiname*`.o` als Standardausgabedatei, anderenfalls `d.out`.

`-p` Die angegebene Prozess-ID *PID* erfassen, ihre Symboltabellen zwischenspeichern und anschließend Vorgang beenden. Wenn die Option `-p` in der Befehlszeile mehrmals vorhanden ist, wird `dtrace` nach dem Ende aller Befehle ebenfalls beendet und meldet den Beendigungsstatus für jeden Prozess. Die erste Prozess-ID wird jedem in der Befehlszeile oder mit der Option `-s` angegebene D-Programm über die Makrovariable `$target` verfügbar gemacht. Weitere Informationen zu Makrovariablen finden Sie in Kapitel 15, „Scripting“.

`-P` Name des zu verfolgenden oder aufzulistenden (Option `-l`) Providers angeben. Die übrigen Prüfpunktbeschreibungsfelder, Modul, Funktion und Name, bleiben leer und treffen, unabhängig von ihren jeweiligen Werten in diesen Feldern, auf alle Prüfpunkte zu. Dem `-P`-Argument kann eine optionale D-Prüfpunktklausel angefügt werden. Die Option `-P` darf in der Befehlszeile mehrmals gleichzeitig verwendet werden.

`-q` Quiet-Modus setzen. `dtrace` unterdrückt Meldungen wie zum Beispiel die Anzahl der mit den angegebenen Optionen und D-Programmen übereinstimmenden Prüfpunkte, und es werden weder Spaltenüberschriften, CPU-ID noch Prüfpunkt-ID ausgegeben noch Zeilenanfangszeichen in die Ausgabe eingefügt. Nur die mit D-Programmanweisungen wie `trace()` und `printf()` verfolgten und formatierten Daten werden an `stdout` ausgegeben.

`-s` Angegebene D-Programmquelldatei kompilieren. Wenn die Option `-e` vorhanden ist, wird das Programm kompiliert, aber keine Instrumentation aktiviert. Wenn die

Option `-l` vorhanden ist, wird das Programm kompiliert und die übereinstimmenden Prüfpunkte werden aufgelistet, aber keine Instrumentation wird aktiviert. Wenn weder `-e` noch `-l` vorhanden sind, wird die mit dem D-Programm angegebene Instrumentation aktiviert und die Ablaufverfolgung beginnt.

`-S` Zwischencode des D-Compilers zeigen. Der D-Compiler erstellt einen Bericht über den für jedes D-Programm generierten Zwischencode und gibt ihn an `stderr` aus.

`-U` Beim Aufruf von cpp(1) (mit der Option `-C` aktiviert) die Definition des angegebenen *Namens* auflösen. Diese Option übergibt jedem Aufruf von `cpp` die Option `-U`.

`-v` Ausführlichen Modus setzen. Wenn die Option `-v` angegeben ist, erstellt `dtrace` einen Programmstabilitätsbericht, aus dem die niedrigste Schnittstellenstabilität und Abhängigkeitsstufe für die angegebenen D-Programme hervorgeht. DTrace-Stabilitätsstufen werden in Kapitel 39, „Stabilität“ ausführlich erläutert.

`-V` Die höchste von `dtrace` unterstützte Version der D-Programmierschnittstelle melden. Die Versionsinformationen werden an `stdout` ausgegeben und der Befehl `dtrace` wird beendet. Weitere Informationen zu Versionsleistungsmerkmalen von DTrace finden Sie in Kapitel 41, „Versionsverwaltung“.

`-w` Destruktive Aktionen in D-Programmen zulassen, die mit den Optionen `-s`, `-P`, `-m`, `-f`, `-n` oder `-i` angegeben wurden. Wenn die Option `-w` nicht angegeben ist, lässt `dtrace` die Kompilierung oder Aktivierung eines D-Programms, das destruktive Aktionen enthält, nicht zu. Destruktive Aktionen werden ausführlicher in Kapitel 10, „Aktionen und Subroutinen“ erläutert.

`-x` Eine DTrace-Laufzeitoption oder D-Compiler-Option aktivieren oder modifizieren. Die Optionen sind in Kapitel 16, „Optionen und Tunables “ aufgeführt. Boolesche Optionen werden durch Angabe ihres Namens aktiviert. Optionen mit Werten werden mit dem Optionsnamen und dem Wert, getrennt durch ein Gleichheitsszeichen (`=`) angegeben.

`-X` Grad der Konformität mit dem ISO-C-Standard angeben, der beim Aufruf von cpp(1) (mit der Option `-C` aktiviert) gewählt werden soll. Das Argument der Option `-X` beeinflusst je nach dem Wert des Argumentbuchstabens den Wert und die Präsenz der `__STDC__`-Makro:

`a` (Standard)	ISO C plus K&R-Kompatibilitätserweiterungen mit semantischen Änderungen nach ISO C. Dies ist der Standardmodus, wenn `-X` nicht angegeben ist. Die vordefinierte Makro `__STDC__` besitzt den Wert 0, wenn `cpp` in Verbindung mit der Option `-Xa` aufgerufen wird.
`c` (Konform)	Strikt ISO-C-konform ohne K&R C-Kompatibilitätserweiterungen. Die vordefinierte Makro

__STDC__ besitzt den Wert 1, wenn cpp in Verbindung mit der Option -Xc aufgerufen wird.

s (K&R C)	Nur K&R C. Die Makro __STDC__ ist nicht definiert, wenn cpp in Verbindung mit der Option -Xs aufgerufen wird.
t (Zwischenstufe)	ISO C plus K&R C-Kompatibilitätserweiterungen ohne die von ISO C geforderten semantische Änderungen. Die vordefinierte Makro __STDC__ besitzt den Wert 0, wenn cpp in Verbindung mit der Option - Xt aufgerufen wird.

Da sich die Option -X nur darauf auswirkt, wie der D-Compiler den C-Preprozessor aufruft, sind die Optionen -Xa und -Xt aus der Sicht von D gleichwertig. Beide Optionen stehen zur Erleichterung der Weiterverwendung von Einstellungen aus C-Build-Umgebungen zur Verfügung.

Unabhängig vom -X-Modus werden die folgenden zusätzlichen C-Preprozessordefinitionen immer angegeben und sind immer in allen Modi gültig:

- __sun
- __unix
- __SVR4
- __sparc (nur auf SPARC®-Systemen)
- __sparcv9 (nur auf SPARC®-Systemen bei der Kompilierung von 64-Bit-Programmen)
- __i386 (nur auf x86-Systemen bei der Kompilierung von 32-Bit-Programmen)
- __amd64 (nur auf x86-Systemen bei der Kompilierung von 64-Bit-Programmen)
- __*'uname -s'*_*'uname -r'*, ersetzt den Dezimalpunkt in der Ausgabe von uname mit einem Unterstrich (_), wie in __SunOS_5_10
- __SUNW_D=1
- __SUNW_D_VERSION=41x*MMmmmuuu* (wobei *MM* der Wert der Hauptversion in hexadezimaler Form, *mmm* der Wert der Unterversion in hexadezimaler Form und *uuu* der Wert der Micro-Version in hexadezimaler Form ist. Nähere Informationen zu DTrace-Versionen finden Sie in Kapitel 41, „Versionsverwaltung“.)

-Z Prüfpunktbeschreibungen zulassen, die auf keinen Prüfpunkt zutreffen. Wenn die Option -Z nicht angegeben ist, meldet dtrace einen Fehler und wird beendet, falls etwaige in D-Programmdateien (Option -s) oder in der Befehlszeile (Optionen -P, -m, -f, -n oder -i) angegebene Prüfpunktbeschreibungen Vereinbarungen enthalten, die auf keinen bekannten Prüfpunkt zutreffen.

Operanden

In der `dtrace`-Befehlszeile dürfen null oder mehr zusätzliche Argumente zur Definition eines Satzes Makrovariablen (`$1`, `$2` usw.) angegeben werden, die in den mit der Option `-s` oder in der Befehlszeile angegebenen D-Programmen verwendet werden können. Die Verwendung von Makrovariablen wird in Kapitel 15, „Scripting“ ausführlicher beschrieben.

Beendigungsstatus

Das Dienstprogramm `dtrace` gibt die folgenden Beendigungswerte zurück:

0 Die angegebenen Anforderungen wurden erfolgreich durchgeführt. Bei D-Programmanforderungen gibt der Beendigungsstatus 0 an, dass die Programme erfolgreich kompiliert, Prüfpunkte erfolgreich aktiviert oder ein anonymer Status erfolgreich abgerufen wurden. `dtrace` gibt auch dann 0 zurück, wenn die angegebenen Ablaufverfolgungsanforderungen Fehler oder Auslassungen aufwiesen.

1 Ein schwerwiegender Fehler ist aufgetreten. Bei D-Programmanforderungen bedeutet der Beendigungsstatus 1, dass eine Programmkompilierung fehlgeschlagen ist oder die angegebene Anforderung nicht erfüllt werden konnte.

2 Es wurden ungültige Befehlszeilenoptionen oder -argumente angegeben.

◆ ◆ ◆ KAPITEL 15

15 Scripting

Das Dienstprogramm dtrace(1M) bietet Ihnen die Möglichkeit, Shell-Skripten ähnliche Interpreterdateien aus D-Programmen zu erstellen, die Sie installieren und als wieder verwendbare, interaktive DTrace-Tools nutzen können. Der D-Compiler und der Befehl `dtrace` stellen einen Satz *Makrovariablen* zur Verfügung, die durch den D-Compiler erweitert werden und das Schreiben von DTrace-Skripten erleichtern. Dieses Kapitel dient als Referenz für die Makrovariableneinrichtung und enthält Tipps zum Erstellen dauerhafter Skripten.

Interpreterdateien

Ähnlich wie die Shell und Dienstprogramme wie awk(1) und perl(1) können mit dtrace(1M) ausführbare Interpreterdateien erzeugt werden. Eine Interpreterdatei beginnt mit einer Zeile in der Form:

`#!` *Pfadname Arg*

wobei *Pfadname* der Pfad des Interpreters und *Arg* ein einzelnes, optionales Argument ist. Wenn eine Interpreterdatei ausgeführt wird, ruft das System den angegebenen Interpreter auf. Wenn *Arg* in der Interpreterdatei angegeben wurde, wird es dem Interpreter als Argument übergeben. Dann werden der Pfad zur Interpreterdatei selbst und etwaige bei der Ausführung zusätzlich angegebene Argumente an die Interpreter-Argumentliste angehängt. Deshalb müssen Sie DTrace-Interpreterdateien stets mit mindestens drei Argumenten erstellen:

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

Wenn die Interpreterdatei ausgeführt wird, ist das Argument für die Option `-s` folglich der Pfadname der Interpreterdatei selbst. Anschließend wird die Datei von `dtrace` gelesen, kompiliert und ausgeführt, als hätten Sie den folgenden Befehl in die Shell eingegeben:

```
# dtrace -s Interpreter-Datei
```

Das folgende Beispiel zeigt, wie eine `dtrace`-Interpreterdatei erzeugt und ausgeführt wird. Schreiben Sie den folgenden D-Quellcode und speichern Sie ihn in einer Datei namens `interp.d`:

```
#!/usr/sbin/dtrace -s
BEGIN
{
    trace("hello");
    exit(0);
}
```

Kennzeichnen Sie die Datei `interp.d` als ausführbar und führen Sie sie wie folgt aus:

```
# chmod a+rx interp.d
# ./interp.d
dtrace: script ’./interp.d’ matched 1 probe
CPU     ID                    FUNCTION:NAME
  1      1                           :BEGIN   hello
#
```

Denken Sie daran, dass die `#!`-Direktive die ersten zwei Zeichen der Datei ohne vorangestellte oder eingefügte Leerstellen enthalten muss. Der D-Compiler ignoriert diese Zeile bei der Verarbeitung der Interpreterdatei automatisch.

`dtrace` verarbeitet Befehlszeilenoptionen mit `getopt(3C)`. Das bedeutet, dass in dem einzelnen Interpreterargument mehrere Optionen kombiniert werden können. Um beispielsweise die Option `-q` in das obige Beispiel einzufügen, ließe sich die Interpreterdirektive wie folgt abändern:

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

Bei der Angabe mehrerer Optionsbuchstaben muss die Option `-s` stets die Liste der booleschen Optionen abschließen, sodass das nächste Argument (der Name der Interpreterdatei) als Argument der Option `-s` verarbeitet wird.

Wenn Sie in der Interpreterdatei mehr als eine Option angeben müssen, die ein Argument benötigt, passen die Optionen und Argumente nicht in das einzelne Interpreter-Argument hinein. Legen Sie die Optionen in diesem Fall mit der Syntax der Direktive `#pragma D option` fest. Für alle `dtrace`-Befehlszeilenoptionen stehen Ihnen `#pragma`-Pendants zur Verfügung, die in Kapitel 16, „Optionen und Tunables “ aufgeführt sind.

Makrovariablen

Der D-Compiler definiert einen Satz integrierter Makrovariablen, auf die Sie beim Schreiben von D-Programmen oder Interpreterdateien zurückgreifen können. Makrovariablen sind IDs mit einem vorangestellten Dollarzeichen ($), die bei der Verarbeitung der Eingabedatei einmal vom D-Compiler ersetzt werden. Der D-Compiler bietet die folgenden Makrovariablen:

TABELLE 15–1 D-Makrovariablen

Name	Beschreibung	Referenz
`$[0-9]+`	Makroargumente	Lesen Sie dazu „Makroargumente“ auf Seite 194
`$egid`	Effektive Gruppen-ID	`getegid(2)`
`$euid`	Effektive Benutzer-ID	`geteuid(2)`
`$gid`	Tatsächliche Gruppen-ID	`getgid(2)`
`$pid`	Prozess-ID	`getpid(2)`
`$pgid`	Prozess-Gruppen-ID	`getpgid(2)`
`$ppid`	ID des übergeordneten Prozesses	`getppid(2)`
`$projid`	Projekt-ID	`getprojid(2)`
`$sid`	Sitzungs-ID	`getsid(2)`
`$target`	ID des Zielprozesses	Lesen Sie dazu „ID des Zielprozesses“ auf Seite 196
`$taskid`	Vorgangs-ID	`gettaskid(2)`
`$uid`	Tatsächliche Benutzer-ID	`getuid(2)`

Außer den Makroargumenten `$[0-9]+` und der Makrovariable `$target` werden alle Makrovariablen durch die ganzzahligen Werte der Systemattribute wie beispielsweise Prozess-ID oder Benutzer-ID ersetzt. Die Variablen werden durch den Attributwert ersetzt, der entweder zu dem aktuellen `dtrace`-Prozess selbst oder dem den D-Compiler ausführenden Prozess gehört.

Durch den Einsatz von Makrovariablen in Interpreterdateien haben Sie die Möglichkeit, dauerhafte D-Programme zu erstellen, die Sie nicht bei jeder Verwendung neu bearbeiten müssen. Wenn Sie beispielsweise alle Systemaufrufe außer den vom Befehl `dtrace` ausgeführten zählen möchten, können Sie die folgende D-Programmklausel mit `$pid` verwenden:

```
syscall::::entry
/pid != $pid/
{
    @calls = count();
}
```

Diese Klausel bringt immer das gewünschte Ergebnis, obwohl jeder Aufruf des Befehls `dtrace` eine andere Prozess-ID haben wird.

Makrovariablen können überall dort in einem D-Programm verwendet werden, wo Ganzzahlen, IDs oder Zeichenketten erlaubt sind. Sie werden nur einmal (d. h. nicht rekursiv) bei der Analyse der Eingabedatei ersetzt. Alle Makrovariablen werden durch separate Eingabesymbole ersetzt und können nicht mit zusätzlichem Text verkettet werden, um ein einzelnes Symbol zu bilden. Wenn beispielsweise `$pid` durch den Wert 456 ersetzt wird, dann wird der D-Code:

```
123$pid
```

nicht als das einzelne, ganzzahlige Symbol 123456, sondern als zwei nebeneinander stehende Symbole 123 und 456 gewertet und ergibt einen Syntaxfehler.

Makrovariablen werden ersetzt und mit angrenzendem Text innerhalb von D-Prüfpunktbeschreibungen am Anfang der Programmklauseln verkettet. In der folgenden Klausel wird beispielsweise mit dem DTrace-Provider `pid` der Befehl `dtrace` instrumentiert:

```
pid$pid:libc.so:printf:entry
{
    ...
}
```

Makrovariablen werden nur einmal in jedem Prüfpunktbeschreibungsfeld ersetzt; sie dürfen keine Begrenzungszeichen für Prüfpunktbeschreibungen (`:`) enthalten.

Makroargumente

Der D-Compiler bietet auch einen Satz Makrovariablen für alle zusätzlichen als Teil des `dtrace`-Befehlsaufrufs angegebenen Argument-Operanden. Auf diese *Makroargumente* wird über die integrierten Namen zugegriffen: `$0` für den Namen der D-Programmdatei oder des `dtrace`-Befehls, `$1` für den ersten zusätzlichen Operanden, `$2` für den zweiten Operanden und so weiter. Wenn Sie die `dtrace`-Option `-s` verwenden, wird `$0` durch den Wert des Namens der mit dieser Option verwendeten Eingabedatei ersetzt. Bei D-Programmen, die in der Befehlszeile angegeben werden, nimmt `$0` den Wert von `argv[0]` an, das zum Ausführen von `dtrace` selbst verwendet wurde.

Makroargumente können in Abhängigkeit von der Form des zugehörigen Texts durch Ganzzahlen, IDs oder Zeichenketten ersetzt werden. Wie alle Makrovariablen können auch

Makroargumente über all dort in einem D-Programm verwendet werden, wo Symbole in Form von Ganzzahlen, IDs und Zeichenketten zulässig sind. Alle nachfolgenden Beispiele könnten gültige D-Ausdrücke bilden, geeignete Makroargumentwerte vorausgesetzt:

```
execname == $1    /* with a string macro argument */
x += $1           /* with an integer macro argument */
trace(x->$1)      /* with an identifier macro argument */
```

Makroargumente können zum Erstellen von `dtrace`-Interpreterdateien eingesetzt werden, die als regelrechte Solaris-Befehle fungieren und Informationen verwenden, die von einem Benutzer oder einem anderen Tool angegeben werden, um deren Verhalten zu modifizieren. So zeichnet beispielsweise die folgende D-Interpreterdatei die durch einen bestimmten Prozess (ID) ausgeführten `write(2)`-Systemaufrufe auf:

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

syscall::write:entry
/pid == $1/
{
}
```

Wenn Sie diese Interpreterdatei als ausführbar kennzeichnen, können Sie der Interpreterdatei den Wert von `$1` über ein zusätzliches Befehlszeilenargument mitteilen:

```
# chmod a+rx ./tracewrite
# ./tracewrite 12345
```

Der entstehende Befehlsaufruf zählt alle durch den Prozess mit der ID 12345 ausgeführten `write(2)`-Systemaufrufe.

Verweist Ihr D-Programm auf ein nicht in der Befehlszeile angegebenes Makroargument, wird eine entsprechende Fehlermeldung ausgegeben und die Programmkompilierung schlägt fehl:

```
# ./tracewrite
dtrace: failed to compile script ./tracewrite: line 4:
  macro argument $1 is not defined
```

Wurde die Option `defaultargs` gesetzt, ist die Referenzierung nicht angegebener Makroargumente durch das Programm möglich. Wenn `defaultargs` gesetzt ist, nehmen die nicht angegebenen Argumente den Wert `0` an. Weitere Informationen zu den Optionen des D-Compilers finden Sie in Kapitel 16, „Optionen und Tunables “. Der D-Compiler gibt auch dann eine Fehlermeldung aus, wenn in der Befehlszeile zusätzliche Argumente angegeben sind, auf die das D-Programm nicht verweist.

Die Werte der Makroargumente müssen mit der Form einer Ganzzahl, ID oder Zeichenkette übereinstimmen. Stimmt das Argument mit keiner dieser Formen überein, meldet der D-Compiler einen entsprechenden Fehler. Wenn Sie einer DTrace-Interpreterdatei ein

Zeichenketten-Makroargument übergeben, schließen Sie das Argument in ein zusätzliches Paar einfacher Anführungszeichen ein, damit die Shell nicht den Inhalt der doppelten Anführungszeichen und der Zeichenkette interpretiert:

```
# ./foo '"a string argument"'
```

Wenn die D-Makroargumente als Zeichenketten-Symbole interpretiert werden sollen, obwohl sie mit der Form einer Ganzzahl oder ID übereinstimmen, stellen Sie dem Namen der Makrovariable oder des Arguments zwei Dollarzeichen voran (z. B. `$$1`). Dadurch wird der D-Compiler gezwungen, die Argumentwerte wie eine in doppelten Anführungszeichen eingeschlossene Zeichenkette zu interpretieren. Alle in D üblichen Ersatzdarstellungen für Zeichenketten (siehe Tabelle 2–5) werden innerhalb von etwaigen Zeichenketten-Makroargumenten aufgelöst. Dabei spielt es keine Rolle, ob sie in der Makroform $*Arg* oder `$$`Arg referenziert werden. Wenn die Option `defaultargs` gesetzt ist, nehmen nicht angegebene Argumente, auf die in der Form `$$`*Arg* verwiesen wird, den Wert der leeren Zeichenkette ("") an.

ID des Zielprozesses

Die Makrovariable `$target` dient zum Erstellen von Skripten, die sich gezielt auf einen Benutzerprozess von Interesse anwenden lassen, der entweder in der `dtrace`-Befehlszeile mit der Option `-p` ausgewählt oder mit der Option `-c` erzeugt wurde. In der Befehlszeile oder mit der Option `-s` angegebene D-Programme werden kompiliert, *nachdem* Prozesse erzeugt oder erfasst (grab) werden, und die Variable `$target` wird durch die ganzzahlige Prozess-ID des ersten dieser Prozesse ersetzt. Mit dem folgenden D-Skript ließe sich beispielsweise die Verteilung der von einem bestimmten Prozess ausgeführten Systemaufrufe ermitteln:

```
syscall:::entry
/pid == $target/
{
    @[probefunc] = count();
}
```

Um die Anzahl der vom Befehl `date(1)` ausgeführten Systemaufrufe zu ermitteln, speichern Sie das Skript in der Datei `syscall.d` und führen folgenden Befehl aus:

```
# dtrace -s syscall.d -c date
dtrace: script 'syscall.d' matched 227 probes
Fri Jul 30 13:46:06 PDT 2004
dtrace: pid 109058 has exited

  gtime                                                             1
  getpid                                                            1
  getrlimit                                                         1
  rexit                                                             1
```

```
ioctl                                                  1
resolvepath                                            1
read                                                   1
stat                                                   1
write                                                  1
munmap                                                 1
close                                                  2
fstat64                                                2
setcontext                                             2
mmap                                                   2
open                                                   2
brk                                                    4
```

◆ ◆ ◆ KAPITEL 16

16 Optionen und Tunables

Zugunsten der Anpassungsfähigkeit bietet DTrace unterschiedliche Flexibilitätsstufen. Um die Notwendigkeit spezifischer Abstimmungen zu minimieren, wurde DTrace mit vernünftigen Standardwerten und flexiblen Standardrichtlinien implementiert. Trotzdem kann es vorkommen, dass das Verhalten von DTrace auf Basis der einzelnen Verbraucher abgestimmt werden muss. In diesem Kapitel werden die DTrace-Optionen und -Tunables (abstimmbare Optionen) sowie die Schnittstellen beschrieben, über die sie modifiziert werden können.

Verbraucheroptionen

Sie passen DTrace an, indem Sie Optionen setzen oder aktivieren. Die verfügbaren Optionen sind in der nachfolgenden Tabelle beschrieben. Für einige Optionen bietet `dtrace`(1M) eine entsprechende Befehlszeilenoption.

TABELLE 16–1 DTrace-Verbraucheroptionen

Optionsname	Wert	`dtrace`(1M)-Alias	Beschreibung	Siehe Kapitel
`aggrate`	*Zeit*		Leserate für Aggregate	Kapitel 9, „Aggregate“
`aggsize`	*Größe*		Größe des Aggregatpuffers	Kapitel 9, „Aggregate“
`bufresize`	`auto` oder `manual`		Richtlinie zur Puffergröße-nänderung	Kapitel 11, „Puffer und Pufferung“
`bufsize`	*Größe*	`-b`	Hauptpuffergröße	Kapitel 11, „Puffer und Pufferung“

TABELLE 16–1 DTrace-Verbraucheroptionen *(Fortsetzung)*

Optionsname	Wert	`dtrace`(1M)-Alias	Beschreibung	Siehe Kapitel
`cleanrate`	*Zeit*		Bereinigungs-häufigkeit. Muss in Anzahl pro Sekunde mit dem Suffix `hz` angegeben werden.	Kapitel 13, „Spekulative Ablaufverfolgung“
`cpu`	*Skalar*	`-c`	CPU, auf der die Ablaufverfolgung aktiviert werden soll	Kapitel 11, „Puffer und Pufferung“
`defaultargs`	—		Verweise auf nicht angegebene Makroargumente zulassen	Kapitel 15, „Scripting“
`destructive`	—	`-w`	Destruktive Aktionen zulassen	Kapitel 10, „Aktionen und Subroutinen“
`dynvarsize`	*Größe*		Speicherplatz für dynamische Variablen	Kapitel 3, „Variablen“
`flowindent`	—	`-F`	Funktionseintritt einrücken und `->` voranstellen; Funktionsrückkehr ausrücken und `<-` voranstellen	Kapitel 14, „Das Dienstprogramm `dtrace`(1M)“
`grabanon`	—	`-a`	Anonymen Status fordern	Kapitel 36, „Anonyme Ablaufverfolgung“
`jstackframes`	*Skalar*		Anzahl der Standard-Stack-Frames für `jstack()`	Kapitel 10, „Aktionen und Subroutinen“
`jstackstrsize`	*Skalar*		Standard-Zeichen-kettenspeicherplatz für `jstack()`	Kapitel 10, „Aktionen und Subroutinen“
`nspec`	*Skalar*		Anzahl der Spekulationen	Kapitel 13, „Spekulative Ablaufverfolgung“

TABELLE 16–1 DTrace-Verbraucheroptionen *(Fortsetzung)*

Optionsname	Wert	dtrace(1M)-Alias	Beschreibung	Siehe Kapitel
quiet	—	-q	Nur ausdrücklich verfolgte Daten ausgeben	Kapitel 14, „Das Dienstprogramm dtrace(1M)“
specsize	*Größe*		Größe des Spekulationspuffers	Kapitel 13, „Spekulative Ablaufverfolgung“
strsize	*Größe*		Zeichenkettengröße	Kapitel 6, „Zeichenketten“
stackframes	*Skalar*		Anzahl der Stack-Frames	Kapitel 10, „Aktionen und Subroutinen“
stackindent	*Skalar*		Anzahl der Leerstellenzeichen für Einrückung der stack()- und ustack()-Ausgabe	Kapitel 10, „Aktionen und Subroutinen“
statusrate	*Zeit*		Statusüberprüfungsrate	
switchrate	*Zeit*		Pufferwechselrate	Kapitel 11, „Puffer und Pufferung“
ustackframes	*Skalar*		Anzahl der Benutzer-Stack-Frames	Kapitel 10, „Aktionen und Subroutinen“

Werte, die eine Größe darstellen, dürfen durch eines der optionalen Suffixe k, m, g oder t ergänzt werden, um die Einheit Kilobyte, Megabyte, Gigabyte bzw. Terabyte anzugeben. Werten, die Zeiten darstellen, kann eines der optionalen Suffixe ns, us, ms, s oder hz angefügt werden, die für Nanosekunden, Mikrosekunden, Millisekunden, Sekunden bzw. Anzahl pro Sekunde stehen.

Modifizieren von Optionen

Optionen können in einem D-Skript anhand von #pragma D, gefolgt von der Zeichenkette option und dem Optionsnamen gesetzt werden. Bei Optionen, die einen Wert annehmen, setzen Sie hinter den Optionsnamen ein Gleichheitszeichen (=) und den Optionswert. Bei allen folgenden Beispielen handelt es sich um gültige Optionseinstellungen:

```
#pragma D option nspec=4
#pragma D option grabanon
#pragma D option bufsize=2g
```

```
#pragma D option switchrate=10hz
#pragma D option aggrate=100us
#pragma D option bufresize=manual
```

Der Befehl·dtrace(1M) akzeptiert Optionseinstellungen auch in der Befehlszeile als Argument für die Option -x. Beispiel:

```
# dtrace -x nspec=4 -x grabanon -x bufsize=2g \
    -x switchrate=10hz -x aggrate=100us -x bufresize=manual
```

Bei Angabe einer ungültigen Option weist dtrace darauf hin, dass der Optionsname ungültig ist, und wird beendet:

```
# dtrace -x wombats=25
dtrace: failed to set option -x wombats: Invalid option name
#
```

Ebenso gibt dtrace eine Meldung aus, wenn ein Optionswert für die angegebene Option ungültig ist:

```
# dtrace -x bufsize=100wombats
dtrace: failed to set option -x bufsize: Invalid value for specified option
#
```

Wird eine Option mehrmals gesetzt, wird die ältere Einstellung durch die jeweils neuere überschrieben. Einige Optionen, wie zum Beispiel grabanon, können *nur* gesetzt werden. Durch ihr bloßes Vorhandensein sind sie bereits gesetzt und Sie können sie anschließend nicht aufheben.

Für eine anonyme Aktivierung gesetzte Optionen werden von dem DTrace-Verbraucher, der den anonymen Status fordert, berücksichtigt. Weitere Informationen zum Aktivieren der anonymen Ablaufverfolgung finden Sie in Kapitel 36, „Anonyme Ablaufverfolgung“.

◆ ◆ ◆ KAPITEL 17

17 Der Provider dtrace

Der Provider dtrace stellt verschiedene Prüfpunkte zur Verfügung, die sich auf DTrace selbst beziehen. Diese Prüfpunkte dienen zum Initialisieren des Status vor Beginn der Ablaufverfolgung, Verarbeiten des Status nach Abschluss der Ablaufverfolgung und Behandeln unerwarteter Ausführungsfehler in anderen Prüfpunkten.

Der Prüfpunkt BEGIN

Der Prüfpunkt BEGIN wird vor jedem anderen Prüfpunkt ausgelöst. Bevor nicht alle BEGIN-Klauseln abgeschlossen sind, wird kein anderer Prüfpunkt ausgelöst. Mit diesem Prüfpunkt lässt sich jeder in anderen Prüfpunkten benötigte Status initialisieren. Das folgende Beispiel zeigt, wie mithilfe des Prüfpunkts BEGIN ein assoziativer Vektor initialisiert werden kann, der mmap(2)-Schutzbits Textrepräsentationen zuweist:

```
BEGIN
{
    prot[0] = "---";
    prot[1] = "r--";
    prot[2] = "-w-";
    prot[3] = "rw-";
    prot[4] = "--x";
    prot[5] = "r-x";
    prot[6] = "-wx";
    prot[7] = "rwx";
}

syscall::mmap:entry
{
    printf("mmap with prot = %s", prot[arg2 & 0x7]);
}
```

Der Prüfpunkt BEGIN wird in einem nicht spezifizierten Kontext ausgelöst. Das bedeutet, dass die Ausgabe von stack() oder ustack() sowie der Wert kontextspezifischer Variablen (z. B.

execname) beliebig sind. Diese Werte sind nicht zuverlässig und von ihrer Interpretation sollten keine wichtigen Informationen abgeleitet werden. Für den Prüfpunkt BEGIN sind keine Argumente definiert.

Der Prüfpunkt END

Der Prüfpunkt END wird nach allen anderen Prüfpunkten ausgelöst. Bevor nicht alle anderen Prüfpunktklauseln abgeschlossen sind, wird dieser Prüfpunkt nicht ausgelöst. Dieser Prüfpunkt kann zum Verarbeiten von abgerufenen Statusinformationen oder zum Formatieren der Ausgabe verwendet werden. Deshalb finden wir im Prüfpunkt END häufig die Aktion printa. () Mit der Kombination aus den Prüfpunkten BEGIN und END lässt sich messen, wie viel Zeit insgesamt auf eine Ablaufverfolgung aufgewendet wird:

```
BEGIN
{
    start = timestamp;
}

/*
 * ... other tracing actions...
 */

END
{
    printf("total time: %d secs", (timestamp - start) / 1000000000);
}
```

Unter „Datennormalisierung“ auf Seite 127 und „printa()“ auf Seite 169 sind weitere übliche Verwendungszwecke des Prüfpunkts END beschrieben.

Ebenso wie für den Prüfpunkt BEGIN sind auch für END keine Argumente definiert. Der Kontext, in dem der Prüfpunkt END ausgelöst wird, ist beliebig und sollte nicht als verlässlich betrachtet werden.

Bei einer Ablaufverfolgung, für die die Option bufpolicy auf fill gesetzt ist, wird genügend Speicherplatz für etwaige Aufzeichnungen aus dem Prüfpunkt END reserviert. Ausführliche·Informationen finden Sie unter „Die Richtlinie fill und END-Prüfpunkte“ auf Seite 159.

Hinweis – Die Aktion `exit()` bewirkt, dass die Ablaufverfolgung beendet wird und löst den Prüfpunkt `END` aus. Es kommt jedoch zu einer gewissen Verzögerung zwischen dem Aufruf der Aktion `exit()` und der Auslösung des Prüfpunkts `END`. Während dieser Verzögerung wird kein Prüfpunkt ausgelöst. Nachdem ein Prüfpunkt die Aktion `exit()` aufgerufen hat, wird der Prüfpunkt `END` erst dann ausgelöst, wenn der DTrace-Verbraucher feststellt, dass `exit()` aufgerufen wurde und die Ablaufverfolgung beendet. Mit der Option `statusrate` lässt sich festlegen, mit welcher Frequenz der Beendigungsstatus überprüft wird. Weitere Informationen finden Sie in Kapitel 16, „Optionen und Tunables “.

Der Prüfpunkt ERROR

Der Prüfpunkt `ERROR` wird ausgelöst, wenn bei der Ausführung einer Klausel für einen DTrace-Prüfpunkt ein Laufzeitfehler auftritt. Wenn beispielsweise mit einer Klausel versucht wird, einen `NULL`-Zeiger zu dereferenzieren, wird der Prüfpunkt `ERROR`, wie das nächste Beispiel veranschaulicht, ausgelöst.

BEISPIEL 17–1 `error.d`: Aufzeichnung von Fehlern

```
BEGIN
{
    *(char *)NULL;
}

ERROR
{
    printf("Hit an error!");
}
```

Wenn Sie dieses Programm ausführen, erhalten Sie eine Ausgabe wie die folgende:

```
# dtrace -s ./error.d
dtrace: script './error.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  2      3                           :ERROR Hit an error!
dtrace: error on enabled probe ID 1 (ID 1: dtrace:::BEGIN): invalid address
(0x0) in action #1 at DIF offset 12
dtrace: 1 error on CPU 2
```

Die Ausgabe zeigt, dass der Prüfpunkt `ERROR` ausgelöst wurde und veranschaulicht, wie `dtrace(1M)` den Fehler meldet. `dtrace` besitzt eine eigene Aktivierung des Prüfpunkts `ERROR`, die es ihm ermöglicht, Fehler zu melden. In Verbindung mit dem Prüfpunkt `ERROR` können Sie eine benutzerdefinierte Fehlerbehandlung einrichten.

Die Argumente für den Prüfpunkt `ERROR` lauten:

`arg1`	Die EPID (ID des aktivierten Prüfpunkts) des Prüfpunkts, der den Fehler verursacht hat
`arg2`	Der Index der Aktion, die den Fehler verursacht hat
`arg3`	Der DIF-Versatz innerhalb dieser Aktion oder `-1`, wenn nicht zutreffend
`arg4`	Der Fehlertyp
`arg5`	Für den Fehlertyp spezifischer Wert

Die nachfolgende Tabelle beschreibt die verschiedenen Fehlertypen und den Wert, den `arg5` je Typ annimmt:

Wert `arg4`	**Beschreibung**	**Bedeutung** `arg5`
`DTRACEFLT_UNKNOWN`	Unbekannter Fehlertyp	Keinen
`DTRACEFLT_BADADDR`	Zugriff auf nicht zugeordnete oder ungültige Adresse	Adresse, auf die zugegriffen wurde
`DTRACEFLT_BADALIGN`	Zugriff auf nicht ausgerichteten Speicher	Adresse, auf die zugegriffen wurde
`DTRACEFLT_ILLOP`	Unzulässiger oder ungültiger Vorgang	Keinen
`DTRACEFLT_DIVZERO`	Division einer Ganzzahl durch Null	Keinen
`DTRACEFLT_NOSCRATCH`	Nicht genügend Scratch-Platz für Scratch-Speicherzuweisung	Keinen
`DTRACEFLT_KPRIV`	Versuchter Zugriff auf eine Kerneladresse oder -Eigenschaft ohne ausreichende Rechte	Adresse, auf die zugegriffen wurde, oder `0`, wenn nicht zutreffend
`DTRACEFLT_UPRIV`	Versuchter Zugriff auf eine Benutzeradresse oder -Eigenschaft ohne ausreichende Rechte	Adresse, auf die zugegriffen wurde, oder `0`, wenn nicht zutreffend
`DTRACEFLT_TUPOFLOW`	DTrace-interner Parameter-Stacküberlauf	Keinen

Wenn die im Prüfpunkt ERROR selbst durchgeführten Aktionen einen Fehler verursachen, wird dieser Fehler kommentarlos verworfen - der Prüfpunkt ERROR wird nicht rekursiv aufgerufen.

Stabilität

Der Provider `dtrace` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Stable	Stable	Common
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Stable	Stable	Common
Argumente	Stable	Stable	Common

◆ ◆ ◆ KAPITEL 18

Der Provider `lockstat`

Der Provider `lockstat` stellt Prüfpunkte zur Verfügung, die zur Unterscheidung von statistischen Daten über Lock-Contentions oder zur Untersuchung nahezu jedes Aspekts des Sperrverhaltens eingesetzt werden können. Der Befehl `lockstat(1M)` ist eigentlich ein DTrace-Verbraucher, der den Provider `lockstat` zum Abrufen der Rohdaten benutzt.

Überblick

Der Provider `lockstat` stellt zwei Arten von Prüfpunkten zur Verfügung: „contention-event-Prüfpunkte“ und „hold-event-Prüfpunkte“.

Contention-event-Prüfpunkte sprechen auf Konkurrenzsituationen (Contentions oder Kollisionen) an einer Synchronisierungsgrundeinheit an und werden ausgelöst, wenn ein Thread gezwungen ist, zu warten, bis eine Ressource verfügbar wird. Solaris ist allgemein für den konkurrenzfreien Betrieb (also ohne Contentions) optimiert. Länger andauernde Konkurrenzsituationen sind folglich nicht zu erwarten. Diese Prüfpunkte helfen Ihnen, Fälle zu verstehen, in welchen es trotzdem zu Konkurrenz kommt. Da Konkurrenzsituationen relativ selten sind, beeinträchtigt die Aktivierung von contention-event-Prüfpunkten die Leistung in der Regel kaum.

Hold-event-Prüfpunkte sprechen auf das Erhalten, Freigeben oder eine andere Art der Manipulation von Synchronisierungsgrundeinheiten an. Diese Prüfpunkte können helfen, beliebige Fragen über die Art und Weise der Manipulation von Synchronisierungsgrundeinheiten zu beantworten. Da in Solaris sehr häufig Synchronisierungsgrundeinheiten erhalten und freigegeben werden (auf belasteten Systemen bewegen wir uns hier in einem Größenbereich von mehreren Millionen Mal pro Sekunde und CPU), bewirkt die Aktivierung von hold-event-Prüfpunkten eine wesentlich höheren Prüfaktivität als die Aktivierung von contention-event-Prüfpunkten. Die Aktivität kann zwar beträchtlich ausfallen, ist aber nicht schädlich. Diese Prüfpunkte können trotzdem gefahrlos auf Produktionssystemen aktiviert werden.

Der Provider `lockstat` stellt Prüfpunkte zur Verfügung, die auf die verschiedenen Synchronisierungsgrundeinheiten in Solaris ansprechen. Der Rest dieses Kapitels befasst sich mit diesen Grundeinheiten und den auf sie ansprechenden Prüfpunkten.

Prüfpunkte für adaptive Sperren

Adaptive Sperren (adaptive locks) schützen einen kritischen Abschnitt durch Mutex (gegenseitiger Ausschluss) und können in den meisten Kontexten im Kernel erworben werden. Da für adaptive Sperren nur wenige Kontextbeschränkungen gelten, stellen sie die große Mehrheit der Synchronisierungsgrundeinheiten im Solaris-Kernel dar. Diese Sperren weisen ein in Bezug auf Konkurrenzsituationen adaptives (dynamisches) Verhalten auf: Wenn ein Thread eine belegte adaptive Sperre fordert, stellt er fest, ob der besitzende Thread derzeit auf einer CPU läuft. Wenn der Besitzer auf einer anderen CPU läuft, wird der fordernde Thread in den *Wartezustand* versetzt. Wenn der Besitzer nicht läuft, wird der fordernde Thread *blockiert*.

Die vier lockstat-Prüfpunkte für adaptive Sperren sind in Tabelle 18–1 aufgeführt. `arg0` enthält für jeden Prüfpunkt einen Zeiger auf die `kmutex_t`-Struktur, die den adaptiven Lock darstellt.

TABELLE 18–1 Prüfpunkte für adaptive Sperren

`adaptive-acquire`	Hold-event-Prüfpunkt, der unmittelbar nach dem Erlangen einer adaptiven Sperre ausgelöst wird.
`adaptive-block`	Contention-event-Prüfpunkt, der ausgelöst wird, nachdem ein durch einen belegten, adaptiven Mutex blockierter Thread „aufgewacht“ ist und den Mutex erlangt hat. Wenn beide Prüfpunkte aktiviert sind, wird `adaptive-block` *vor* `adaptive-acquire` ausgelöst. Für das Erlangen einer einzigen Sperre können die Prüfpunkte `adaptive-block` und `adaptive-spin` ausgelöst werden. `arg1` für `adaptive-block` enthält die Schlafzeit in Nanosekunden.
`adaptive-spin`	Contention-event-Prüfpunkt, der ausgelöst wird, nachdem ein durch einen belegten, adaptiven Mutex in den Wartezustand versetzter Thread den Mutex erfolgreich erlangt hat. Wenn beide Prüfpunkte aktiviert sind, wird `adaptive-spin` *vor* `adaptive-acquire` ausgelöst. Für das Erlangen einer einzigen Sperre können die Prüfpunkte `adaptive-block` und `adaptive-spin` ausgelöst werden. `arg1` für `adaptive-spin` enthält die *Spin-Anzahl*: die Zeit in Nanosekunden, die in der Warteschleife vor dem Erlangen der Sperre verbracht wurde.
`adaptive-release`	Hold-event-Prüfpunkt, der unmittelbar nach der Freigabe einer adaptiven Sperre ausgelöst wird.

Spinlock-Prüfpunkte

In einigen Kernel-Kontexten wie beispielsweise Interrupts auf hoher Ebene oder in jedem den Dispatcher-Zustand manipulierenden Kontext können Threads nicht blockiert werden. In diesen Kontexten wird die Verwendung von adaptiven Sperren durch diese Einschränkung verhindert. Stattdessen wird der Mutex in diesen Kontexten anhand von *Spinlocks* ausgesprochen. Das Verhalten dieser Sperren bei Konkurrenz besteht im Durchlaufen einer Warteschleife, bis die Sperre vom Besitzer-Thread freigegeben wird. Die drei Prüfpunkte im Zusammenhang mit Spinlocks sind in Tabelle 18–2 beschrieben.

TABELLE 18–2 Spinlock-Prüfpunkte

`spin-acquire`	Hold-event-Prüfpunkt, der unmittelbar nach dem Erlangen eines Spinlocks ausgelöst wird.
`spin-spin`	Contention-event-Prüfpunkt, der ausgelöst wird, nachdem ein Thread, der durch einen belegten Spinlock in eine Warteschleife gestellt wurde, den Spinlock erfolgreich erlangt hat. Wenn beide Prüfpunkte aktiviert sind, wird `spin-spin` *vor* `spin-acquire` ausgelöst. `arg1` für `spin-spin` enthält die *Spin-Zeit*: die Zeit in Nanosekunden, die im Spin-Status vor dem Erlangen der Sperre verbracht wurde. Die Spin-Anzahl an sich ist nicht sehr bedeutungsvoll, kann aber zum Vergleich von Wartezeiten genutzt werden.
`spin-release`	Hold-event-Prüfpunkt, der unmittelbar nach der Freigabe eines Spinlocks ausgelöst wird.

Adaptive Sperren sind sehr viel häufiger als Spinlocks. Um diese Beobachtung auf Daten zu stützen, greifen wir auf das folgende Skript zurück, das die Gesamtzahl beider Lock-Typen anzeigt.

```
lockstat:::adaptive-acquire
/execname == "date"/
{
    @locks["adaptive"] = count();
}

lockstat:::spin-acquire
/execname == "date"/
{
    @locks["spin"] = count();
}
```

Führen Sie dieses Skript in einem Fenster aus und den Befehl `date(1)` in einem anderen. Wenn Sie das DTrace-Skript beenden, erhalten Sie eine Ausgabe wie im nächsten Beispiel:

```
# dtrace -s ./whatlock.d
dtrace: script ’./whatlock.d’ matched 5 probes
^C
```

```
spin                                                    26
adaptive                                              2981
```

Wie diese Ausgabe zeigt, sind über 99 Prozent der bei der Ausführung des Befehls `date` erlangten Sperren adaptive Sperren. Es überrascht Sie vielleicht, dass bei einer einfachen Aktion wie *date* `so` viele Sperren erlangt werden. Die große Anzahl der Sperren ist ein natürlicher Nebeneffekt der in einem so extrem skalierbaren System wie dem Solaris-Kernel erforderlichen feinkörnigen Sperrstrukturen.

Threadsperren

Threadsperren sind eine spezielle Form des Spinlocks, die dazu dienen, einen Thread zu sperren, damit sein Status geändert werden kann. Während Hold-Ereignisse von Threadsperren als hold-event-Prüfpunkte für Spinlocks zur Verfügung stehen (d. h. als `spin-acquire` und `spin-release`), gibt es für spezifisch auf Threadsperren bezogene Konkurrenzereignisse einen eigenen Prüfpunkt. Der Prüfpunkt für Threadsperren-Hold-Ereignisse ist in Tabelle 18–3 beschrieben.

TABELLE 18–3 Threadsperren-Prüfpunkt

`thread-spin`	Contention-event-Prüfpunkt, der ausgelöst wird, nachdem ein Thread in einer Warteschleife an einer Threadsperre gewartet hat. Wie auch bei anderen contention-event-Prüfpunkten, wird `thread-spin` vor `spin-acquire` ausgelöst, wenn sowohl der contention-event-Prüfpunkt als auch der hold-event-Prüfpunkt aktiviert sind. Im Gegensatz zu anderen contention-event-Prüfpunkten wird thread-spin jedoch *vor* dem tatsächlichen Erlangen der Sperre ausgelöst. So können auf eine `spin-acquire`- mehrere `thread-spin`-Prüfpunktauslösungen kommen.

Prüfpunkte für Leser/Schreiber-Sperren

Leser/Schreiber-Sperren setzen eine Richtlinie um, die entweder mehrere Leser (reader) *oder* einen einzigen Schreiber (writer) in einem kritischen Abschnitt zulässt - nicht aber beides. Diese Sperren kommen in der Regel in Strukturen vor, die häufiger durchsucht als geändert werden und für die im kritischen Abschnitt viel Zeit vorgesehen ist. Wenn nicht viel Zeit im kritischen Abschnitt zur Verfügung steht, werden Leser/Schreiber-Sperren implizit über den für die Implementierung der Sperre genutzten gemeinsamen Speicher serialisiert. Dabei wird ihnen kein Vorrang gegenüber adaptiven Sperren eingeräumt. Unter `rwlock(9F)` finden Sie weitere Informationen zu Leser/Schreiber-Sperren.

Tabelle 18–4 zeigt die Prüfpunkte für Leser/Schreiber-Sperren. `arg0` enthält für jeden Prüfpunkt einen Zeiger auf die `krwlock_t`-Struktur, die den adaptiven Lock darstellt.

TABELLE 18–4 Prüfpunkte für Leser/Schreiber-Sperren

`rw-acquire`	Hold-event-Prüfpunkt, der unmittelbar nach dem Erlangen einer Leser/Schreiber-Sperre ausgelöst wird. `arg1` enthält die Konstante `RW_READER`, wenn die Sperre als Leser erlangt wurde und `RW_WRITER`, wenn sie als Schreiber erlangt wurde.
`rw-block`	Contention-event-Prüfpunkt, der ausgelöst wird, nachdem ein an einer belegten Leser/Schreiber-Sperre blockierter Thread aufgewacht ist und die Sperre erlangt hat. `arg1` enthält die Dauer (in Nanosekunden), die der aktuelle Thread schlafen gelegt wurde, bis er die Sperre erlangen konnte. `arg2` enthält die Konstante `RW_READER`, wenn die Sperre als Leser erlangt wurde und `RW_WRITER`, wenn sie als Schreiber erlangt wurde. `arg3` und `arg4` enthalten weitere Informationen zu den Gründen für die Blockierung. `arg3` ist nicht Null, wenn - und nur dann - die Sperre beim Blockieren des aktuellen Threads als Schreiber belegt war. `arg4` enthält die Anzahl der Leser zum Zeitpunkt der Blockierung des aktuellen Threads. Wenn sowohl der Prüfpunkt `rw-block` als auch `rw-acquire` angegeben sind, wird `rw-block` *vor* `rw-acquire` ausgelöst.
`rw-upgrade`	Hold-event-Prüfpunkt, der ausgelöst wird, nachdem ein Thread eine Leser/Schreiber-Sperre erfolgreich vom Leser- auf den Schreiberstatus heraufgestuft hat. Für diese Upgrades gibt es kein Konkurrenzereignis, da sie nur über eine nicht-blockierende Schnittstelle, `rw_tryupgrade(9F)`, möglich sind.
`rw-downgrade`	Hold-event-Prüfpunkt, der ausgelöst wird, nachdem ein Thread seinen Besitzerstatus für eine Leser/Schreiber-Sperre von Schreiber auf Leser herabgestuft hat. Für diese Downgrades gibt es keine Konkurrenzereignisse, da sie stets ohne Kollision erfolgen.
`rw-release`	Hold-event-Prüfpunkt, der unmittelbar nach der Freigabe einer Leser/Schreiber-Sperre ausgelöst wird. `arg1` enthält die Konstante `RW_READER`, wenn die freigegebene Sperre als Leser belegt wurde und `RW_WRITER`, wenn sie als Schreiber belegt wurde. Aufgrund von Up- und Downgrades wird die Sperre möglicherweise *nicht* in der Form freigegeben, in der sie erhalten wurde.

Stabilität

Der Provider `lockstat` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	Common
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Name	Evolving	Evolving	Common
Argumente	Evolving	Evolving	Common

◆ ◆ ◆ KAPITEL 19

Der Provider `profile`

Der Provider `profile` stellt Prüfpunkte für zeitbasierte Interrupts zur Verfügung, die mit einem festgelegten, angegebenen Intervall ausgelöst werden. Diese *nicht verankerten* Prüfpunkte sind nicht mit einem bestimmten Punkt in der Ausführung, sondern mit dem asynchronen Interrupt-Ereignis verknüpft. Sie dienen zum Prüfen einiger Aspekte des Systemstatus im Abstand von bestimmten Zeiteinheiten. Auf Grundlage der Messdaten lassen sich Rückschlüsse auf das Systemverhalten ziehen. Bei einer hohen Prüfrate oder einer langen Prüfzeit sind genaue Rückschlüsse möglich. In Verbindung mit DTrace-Aktionen können mit dem Provider `profile` praktisch alle Aspekte des Systems geprüft werden. So könnten Sie beispielsweise den Zustand des aktuellen Threads, der CPU oder der aktuellen Maschinenanweisung prüfen.

Hinweis – Thread-lokale Variablen sind für Prüfpunkte aus dem Provider `profile` nicht erreichbar. Die Verwendung des speziellen Bezeichners `self` mit einem solchen Prüfpunkt zur Referenzierung einer thread-lokalen Variable generiert keine Ausgabe.

`profile-`*n*-Prüfpunkte

Ein `profile-`*n*-Prüfpunkt wird in einem festgelegten, regelmäßigen Abstand auf jeder CPU mit einer hohen Interrupt-Ebene ausgelöst. Das Auslösungsintervall des Prüfpunkts wird durch den Wert von *n* bestimmt: Die Interrupt-Quelle wird *n*-mal pro Sekunde ausgelöst. Sie können *n* auch ein optionales Zeitsuffix anfügen. In diesem Fall wird *n* in der mit dem Suffix angegebenen Zeiteinheit interpretiert. Tabelle 19–1 zeigt die gültigen Suffixe und die von ihnen bezeichneten Einheiten.

TABELLE 19–1 Gültige Zeitsuffixe

Suffix	Zeiteinheit
nsec oder ns	Nanosekunden
usec oder us	Mikrosekunden
msec oder ms	Millisekunden
sec oder s	Sekunden
min oder m	Minuten
hour oder h	Stunden
day oder d	Tage
hz	Hertz (Frequenz pro Sekunde)

Das nächste Beispiel erzeugt einen Prüfpunkt, der den aktuell laufenden Prozess prüft und mit einer Frequenz von 97 Hertz ausgelöst wird:

```
#pragma D option quiet

profile-97
/pid != 0/
{
    @proc[pid, execname] = count();
}

END
{
    printf("%-8s %-40s %s\n", "PID", "CMD", "COUNT");
    printa("%-8d %-40s %@d\n", @proc);
}
```

Wenn der obige Code kurz ausgeführt wird, erhalten wir eine ähnliche Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./prof.d
^C
PID      CMD                                      COUNT
223887   sh                                       1
100360   httpd                                    1
100409   mibiisa                                  1
223887   uname                                    1
218848   sh                                       2
218984   adeptedit                                2
100224   nscd                                     3
3        fsflush                                  4
```

```
2        pageout                                 6
100372   java                                    7
115279   xterm                                   7
100460   Xsun                                    7
100475   perfbar                                 9
223888   prstat                                  15
```

Mit dem `profile-`*n*-Provider lassen sich auch Informationen über den laufenden Prozess abrufen. Das folgende D-Beispielskript enthält einen profile-Prüfpunkt mit 1.001 Hertz, der zum Abrufen der aktuellen Priorität des angegebenen Prozesses dient:

```
profile-1001
/pid == $1/
{
    @proc[execname] = lquantize(curlwpsinfo->pr_pri, 0, 100, 10);
}
```

Um das Beispielskript in Aktion zu sehen, geben Sie die folgenden Befehle in ein Fenster ein:

```
$ echo $$
12345
$ while true ; do let i=0 ; done
```

Führen Sie das D-Skript einen kurze Zeit lang in einem anderen Fenster aus. Dabei müssen Sie *12345* mit der PID ersetzen, die Ihr `echo`-Befehl zurückgegeben hat.

```
# dtrace -s ./profpri.d 12345
 dtrace: script ’./profpri.d’ matched 1 probe
^C
ksh
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 7443
              10 |@@@@@@@                                  2235
              20 |@@@@                                     1679
              30 |@@@                                      1119
              40 |@                                        560
              50 |@                                        554
              60 |                                         0
```

Diese Ausgabe zeigt die Verzerrung der Timesharing-Scheduling-Klasse. Da der Shell-Prozess auf der CPU in einer Warteschleife läuft, wird dessen Priorität vom System stetig herabgesetzt. Wenn der Shell-Prozess seltener laufen würde, käme ihm eine höhere Priorität zuteil. Das Ergebnis sehen Sie, wenn Sie in die wartende Shell Strg-C eingeben und das Skript erneut ausführen:

```
# dtrace -s ./profpri.d 494621
 dtrace: script ’./profpri.d’ matched 1 probe
```

Geben Sie nun einige Zeichen in die Shell ein. Nach Beendigung des DTrace-Skripts erhalten Sie eine ähnliche Ausgabe wie in diesem Beispiel:

```
ksh
           value  ------------- Distribution ------------- count
              40 |                                         0
              50 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 14
              60 |                                         0
```

Da der Shell-Prozess nicht in einer Warteschleife auf der CPU gelaufen ist, sondern sich in Erwartung einer Benutzereingabe schlafen gelegt hatte, hat er, *als* er schließlich ausgeführt wurde, eine wesentlich höhere Priorität erhalten.

tick-*n*-Prüfpunkte

Wie `profile-`*n*-Prüfpunkte werden auch `tick-`*n*-Prüfpunkte in einem festgelegten Abstand und mit hoher Interrupt-Ebene ausgelöst. Im Gegensatz zu `profile-`*n*-Prüfpunkten, die auf *jeder* CPU ausgelöst werden, werden `tick-`*n*-Prüfpunkte jedoch pro Intervall nur auf *einer* CPU ausgelöst. Die CPU, um die es sich dabei handelt, kann sich im Lauf der Zeit ändern. Wie bei `profile-`*n*-Prüfpunkten wird *n* standardmäßig als Rate pro Sekunde interpretiert, kann aber auch durch ein optionales Suffix ergänzt werden. `tick-`*n*-Prüfpunkte dienen zu verschiedenen Zwecken, wie etwa regelmäßige Ausgaben zu liefern oder regelmäßige Aktionen durchzuführen.

Argumente

Die Argumente für `profile`-Prüfpunkte lauten:

`arg0`	Der Programmzähler im Kernel zum Zeitpunkt der Prüfpunktauslösung, oder 0, wenn der aktuelle Prozess zum Zeitpunkt der Prüfpunktauslösung nicht im Kernel ausgeführt wurde
`arg1`	Der Programmzähler im Prozess auf Benutzerebene zum Zeitpunkt der Prüfpunktauslösung, oder 0, wenn der aktuelle Prozess zum Zeitpunkt der Prüfpunktauslösung im Kernel ausgeführt wurde

Wie aus den Beschreibungen hervorgeht, ist `arg1` Null, wenn `arg0` nicht Null ist, und umgekehrt ist `arg1` nicht Null, wenn `arg0` Null ist. Folglich können Sie mithilfe von `arg0` und `arg1` wie in diesem einfachen Beispiel zwischen Benutzer- und Kernelebene unterscheiden:

```
profile-1ms
{
    @ticks[arg0 ? "kernel" : "user"] = count();
}
```

Timerauflösung

Der Provider `profile` stützt sich auf Intervall-Timer mit frei wählbarer Auflösung im Betriebssystem. Auf Architekturen, die keine wirklich auf arbiträren Intervallen beruhenden Interrupts unterstützen, ist die Frequenz durch den Systemuhrtakt begrenzt, die von der Kernelvariable `hz` vorgegeben ist. Prüfpunkte mit einer höheren Frequenz als `hz` werden auf diesen Architekturen mehrmals alle 1/`hz` Sekunden ausgelöst. Beispielsweise wird ein `profile`-Prüfpunkt mit 1000 Hertz auf einer solchen Architektur, auf der `hz` auf 100 gesetzt ist, alle zehn Millisekunden zehnmal schnell hintereinander ausgelöst. Auf einer Plattform, die arbiträre Auflösungen unterstützt, würde der `profile`-Prüfpunkt mit 1000 Hertz genau jede Millisekunde einmal ausgelöst werden.

Die Auflösung einer Architektur lässt sich wie folgt testen:

```
profile-5000
{
    /*
     * We divide by 1,000,000 to convert nanoseconds to milliseconds, and
     * then we take the value mod 10 to get the current millisecond within
     * a 10 millisecond window.  On platforms that do not support truly
     * arbitrary resolution profile probes, all of the profile-5000 probes
     * will fire on roughly the same millisecond.  On platforms that
     * support a truly arbitrary resolution, the probe firings will be
     * evenly distributed across the milliseconds.
     */
    @ms = lquantize((timestamp / 1000000) % 10, 0, 10, 1);
}

tick-1sec
/i++ >= 10/
{
    exit(0);
}
```

Auf einer Architektur, die `profile`-Prüfpunkte mit arbiträrer Auflösung unterstützt, ergibt die Ausführung des Beispielskripts eine gleichmäßige Verteilung:

```
# dtrace -s ./restest.d
 dtrace: script './restest.d' matched 2 probes
CPU     ID                    FUNCTION:NAME
  0  33631                      :tick-1sec


           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@                                      10760
               1 |@@@@                                     10842
```

```
               2 |@@@@                                     10861
               3 |@@@                                      10820
               4 |@@@                                      10819
               5 |@@@                                      10817
               6 |@@@@                                     10826
               7 |@@@@                                     10847
               8 |@@@@                                     10830
               9 |@@@@                                     10830
```

Auf einer Architektur, die `profile`-Prüfpunkte mit arbiträrer Auflösung nicht unterstützt, ergibt die Ausführung des Beispielskripts eine ungleichmäßige Verteilung:

```
# dtrace -s ./restest.d
 dtrace: script './restest.d' matched 2 probes
 CPU     ID                    FUNCTION:NAME
  0  28321                      :tick-1sec


           value  ------------- Distribution ------------- count
               4 |                                         0
               5 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  107864
               6 |                                         424
               7 |                                         255
               8 |                                         496
               9 |                                         0
```

Auf diesen Architekturen kann `hz` zur Verbesserung der effektiven Profilauflösung unter `/etc/system` manuell angepasst werden.

Derzeit unterstützen alle Varianten von UltraSPARC (`sun4u`) `profile`-Prüfpunkte mit arbiträrer Auflösung. Auch zahlreiche Varianten der x86-Architektur (`i86pc`) bieten Unterstützung für `profile`-Prüfpunkte mit frei wählbarer Auflösung, einige ältere Varianten jedoch nicht.

Prüfpunkterzeugung

Im Gegensatz zu anderen Providern erzeugt der Provider `profile` die Prüfpunkte dynamisch auf Bedarfsbasis. Der gewünschte profile-Prüfpunkt scheint in einer Auflistung aller Prüfpunkte (z. B. mit `dtrace -l -P profile`) möglicherweise nicht auf, wird aber erzeugt, wenn er ausdrücklich aktiviert wird.

Auf Architekturen, die `profile`-Prüfpunkte mit frei wählbarer Auflösung unterstützen, würde die Maschine durch ein zu kurzes Intervall kontinuierlich zeitbasierte Interrupts abfangen und einen Denial-of-Service-Fehler auslösen. Um dies zu verhindern, lehnt der Provider `profile` kommentarlos die Erzeugung von Prüfpunkten ab, die ein Intervall von weniger als 200 Mikrosekunden bewirken würden.

Stabilität

Der Provider `profile` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	Common
Modul	Unstable	Unstable	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	Common
Argumente	Evolving	Evolving	Common

◆ ◆ ◆ KAPITEL 20

20 Der Provider `fbt`

In diesem Kapitel wird der fbt-Provider (function boundary tracing) besprochen, der Prüfpunkte für den Eintritt in und die Rückkehr aus den meisten Funktionen im Solaris-Kernel zur Verfügung stellt. Die Funktion ist die Grundeinheit des Programmtexts. In einem gut gestalteten System führt jede Funktion eine diskrete und genau definierte Operation an einem angegebenen Objekt oder einer Serie gleicher Objekte durch. Aus diesem Grund stellt FBT selbst auf den kleinsten Solaris-Systemen rund 20.000 Prüfpunkte zur Verfügung.

Ähnlich wie andere DTrace-Provider bewirkt FBT keine Prüfaktivität, wenn er nicht ausdrücklich aktiviert wird. Bei Aktivierung löst FBT nur in den untersuchten Funktionen Prüfaktivität aus. Die FBT-Implementierung ist stark an der jeweiligen Befehlssatzarchitektur ausgerichtet. FBT wurde sowohl auf SPARC- als auch x86-Plattformen implementiert. Jeder Befehlssatz enthält einige wenige Funktionen, die keine anderen Funktionen aufrufen und vom Compiler maximal optimiert werden (so genannte *Leaf-Funktionen*) und nicht von FBT instrumentiert werden können. Für diese Funktionen bietet DTrace keine Prüfpunkte.

Eine effektive Nutzung von FBT-Prüfpunkten setzt die Kenntnis der Betriebssystemimplementierung voraus. Wir empfehlen daher, auf FBT nur zur Entwicklung von Kernelsoftware zurückzugreifen, oder wenn andere Prüfpunkte für den Zweck nicht ausreichen. Mithilfe anderer DTrace-Provider, einschließlich `syscall`, `sched`, `proc` und `io`, lassen sich die meisten Fragen in Bezug auf die Systemanalyse beantworten, ohne dass eine Kenntnis der Betriebssystemimplementierung erforderlich ist.

Prüfpunkte

FBT stellt einen Prüfpunkt an der *Grenze* der meisten Funktionen im Kernel bereit. Die Grenze einer Funktion wird beim Eintritt in die Funktion und bei der Rückkehr von der Funktion übertreten. FBT bietet deshalb zwei Prüfpunkte für jede Funktion im Kernel: eine beim Funktionseintritt und eine bei Rückkehr von der Funktion. Diese Prüfpunkte heißen `entry` bzw. `return`. Als Teil des Prüfpunkts werden der Funktions- und der Modulname angegeben. Alle FBT-Prüfpunkte geben einen Funktions- und einen Modulnamen an.

Prüfpunktargumente

`entry`-Prüfpunkte

Die Argumente für `entry`-Prüfpunkte stimmen mit den Argumenten für die entsprechende Funktion im Betriebssystemkernel überein. Auf die Argumente kann nach Art des Typs mit dem Vektor `args[]` zugegriffen werden. Auf diese Argumente kann in Form von `int64_t` über `arg0` .. `arg`*n*`-Variablen zugegriffen werden.`

`return`-Prüfpunkte

Während eine Funktion nur einen einzigen Eintrittspunkt hat, kann sie an mehreren Punkten zum Aufrufer zurückkehren. In der Regel ist man entweder an dem Wert interessiert, den eine Funktion zurückgibt, oder an der Tatsache, dass sie überhaupt zurückkehrt, weniger jedoch, an dem spezifischen Rückkehrpfad. FBT ruft die verschiedenen Rückkehrpunkte einer Funktion deshalb in einen einzigen `return`-Prüfpunkt ab. Wenn Sie an dem genauen Rückkehrpfad interessiert sind, können Sie den `args[0]`-Wert des `return`-Prüfpunkts untersuchen, der den *Versatz* (Offset) in Byte der rückkehrenden Anweisung im Funktionstext wiedergibt.

Hat die Funktion einen Rückgabewert, ist dieser in `args[1]` gespeichert. Wenn eine Funktion keinen Rückgabewert besitzt, ist `args[1]` nicht definiert.

Beispiele

FBT ermöglicht Ihnen eine einfache Untersuchung der Kernelimplementierung. Das folgende Beispielskript zeichnet die erste `ioctl(2)` eines jeden `xclock`-Prozesses auf und folgt dann dem weiteren Codepfad durch den Kernel:

```
/*
 * To make the output more readable, we want to indent every function entry
 * (and unindent every function return).  This is done by setting the
 * "flowindent" option.
 */
#pragma D option flowindent

syscall::ioctl:entry
/execname == "xclock" && guard++ == 0/
{
    self->traceme = 1;
    printf("fd: %d", arg0);
}
```

```
fbt:::
/self->traceme/
{}

syscall::ioctl:return
/self->traceme/
{
    self->traceme = 0;
    exit(0);
}
```

Die Ausführung dieses Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./xioctl.d
dtrace: script ’./xioctl.d’ matched 26254 probes
CPU FUNCTION
  0  => ioctl                                  fd: 3
  0    -> ioctl
  0      -> getf
  0        -> set_active_fd
  0        <- set_active_fd
  0      <- getf
  0      -> fop_ioctl
  0        -> sock_ioctl
  0          -> strioctl
  0            -> job_control_type
  0            <- job_control_type
  0            -> strcopyout
  0              -> copyout
  0              <- copyout
  0            <- strcopyout
  0          <- strioctl
  0        <- sock_ioctl
  0      <- fop_ioctl
  0      -> releasef
  0        -> clear_active_fd
  0        <- clear_active_fd
  0        -> cv_broadcast
  0        <- cv_broadcast
  0      <- releasef
  0    <- ioctl
  0  <= ioctl
```

Die Ausgabe zeigt, dass ein `xclock`-Prozess `ioctl()` auf einem Dateibezeichner aufgerufen hat, der scheinbar einem Socket entspricht.

Darüber hinaus kann Ihnen FBT dabei helfen, das Verhalten von Kerneltreibern zu verstehen. Beispielsweise kann im Fall des Treibers `ssd(7D)` über zahlreiche Codepfade `EIO`

zurückgegeben werden. Das folgende Beispiel zeigt, dass sich mit FBT problemlos der genaue Codepfad ermitteln lässt, der eine Fehlerbedingung verursacht hat:

```
fbt:ssd::return
/arg1 == EIO/
{
    printf("%s+%x returned EIO.", probefunc, arg0);
}
```

Um weitere Informationen über eine Rückkehr von `EIO` zu erhalten, könnte man alle `fbt`-Prüfpunkte spekulativ verfolgen und anschließend, je nach Rückgabewert der spezifischen Funktion, `commit()` (oder `discard()`) anwenden. Ausführliche Informationen zur spekulativen Ablaufverfolgung finden Sie in Kapitel 13, „Spekulative Ablaufverfolgung“.

Alternativ können Sie FBT einsetzen, um den innerhalb eines angegebenen Moduls aufgerufenen Funktionen auf den Grund zu gehen. Im nächsten Beispiel werden alle im UFS aufgerufenen Funktionen aufgelistet:

```
# dtrace -n fbt:ufs::entry'{@a[probefunc] = count()}'
dtrace: description 'fbt:ufs::entry' matched 353 probes
^C
  ufs_ioctl                                                         1
  ufs_statvfs                                                       1
  ufs_readlink                                                      1
  ufs_trans_touch                                                   1
  wrip                                                              1
  ufs_dirlook                                                       1
  bmap_write                                                        1
  ufs_fsync                                                         1
  ufs_iget                                                          1
  ufs_trans_push_inode                                              1
  ufs_putpages                                                      1
  ufs_putpage                                                       1
  ufs_syncip                                                        1
  ufs_write                                                         1
  ufs_trans_write_resv                                              1
  ufs_log_amt                                                       1
  ufs_getpage_miss                                                  1
  ufs_trans_syncip                                                  1
  getinoquota                                                       1
  ufs_inode_cache_constructor                                       1
  ufs_alloc_inode                                                   1
  ufs_iget_alloced                                                  1
  ufs_iget_internal                                                 2
  ufs_reset_vnode                                                   2
  ufs_notclean                                                      2
  ufs_iupdat                                                        2
  blkatoff                                                          3
```

```
  ufs_close                                                 5
  ufs_open                                                  5
  ufs_access                                                6
  ufs_map                                                   8
  ufs_seek                                                 11
  ufs_addmap                                               15
  rdip                                                     15
  ufs_read                                                 15
  ufs_rwunlock                                             16
  ufs_rwlock                                               16
  ufs_delmap                                               18
  ufs_getattr                                              19
  ufs_getpage_ra                                           24
  bmap_read                                                25
  findextent                                               25
  ufs_lockfs_begin                                         27
  ufs_lookup                                               46
  ufs_iaccess                                              51
  ufs_imark                                                92
  ufs_lockfs_begin_getpage                                102
  bmap_has_holes                                          102
  ufs_getpage                                             102
  ufs_itimes_nolock                                       107
  ufs_lockfs_end                                          125
  dirmangled                                              498
  dirbadname                                              498
```

Wenn Sie den Zweck oder die Argumente einer Kernelfunktion kennen, können Sie mithilfe von FBT nachvollziehen, wie oder weshalb diese Funktion aufgerufen wird. `putnext(9F)` nimmt beispielsweise als erstes Element einen Zeiger auf eine `queue(9S)`-Struktur an. Die Komponente `q_qinfo` der Struktur `queue` ist ein Zeiger auf eine `qinit(9S)`-Struktur. Die Komponente `qi_minfo` der Struktur `qinit`besitzt einen Zeiger auf eine `module_info(9S)`-Struktur, die in ihrer Komponente `mi_idname` den Modulnamen enthält. Im nächsten Beispiel werden diese Informationen zusammengefügt. Dabei werden mit demFBT Prüfpunkt in `putnext`die `putnext(9F)` -Aufrufe nach Modulnamen aufgezeichnet:

```
fbt::putnext:entry
{
    @calls[stringof(args[0]->q_qinfo->qi_minfo->mi_idname)] = count();
}
```

Die Ausführung des obigen Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./putnext.d
^C

  iprb                                                      1
  rpcmod                                                    1
```

```
  pfmod                                                         1
  timod                                                         2
  vpnmod                                                        2
  pts                                                          40
  conskbd                                                      42
  kb8042                                                       42
  tl                                                           58
  arp                                                         108
  tcp                                                         126
  ptm                                                         249
  ip                                                          313
  ptem                                                        340
  vuid2ps2                                                    361
  ttcompat                                                    412
  ldterm                                                      413
  udp                                                         569
  strwhead                                                    624
  mouse8042                                                   726
```

Außerdem lässt sich mit FBT die in einer bestimmten Funktion verbrachte Zeit ermitteln. Aus dem nächsten Beispiel geht hervor, wie sich die Aufrufer der DDI-Verzögerungsroutinen `drv_usecwait`(9F) und `delay`(9F) ermitteln lassen.

```
fbt::delay:entry,
fbt::drv_usecwait:entry
{
    self->in = timestamp
}

fbt::delay:return,
fbt::drv_usecwait:return
/self->in/
{
    @snoozers[stack()] = quantize(timestamp - self->in);
    self->in = 0;
}
```

Besonders interessant ist es, dieses Beispielskript beim Booten auszuführen. Kapitel 36, „Anonyme Ablaufverfolgung“ beschreibt das Vorgehen zum Ausführen einer anonymen Ablaufverfolgung während des Bootens eines Systems. Nach dem Neustart kann eine Ausgabe wie im folgenden Beispiel angezeigt werden:

```
# dtrace -ae

              ataʻata_wait+0x34
              ataʻata_id_common+0xf5
              ataʻata_disk_id+0x20
              ataʻata_drive_type+0x9a
```

```
              ata‘ata_init_drive+0xa2
              ata‘ata_attach+0x50
              genunix‘devi_attach+0x75
              genunix‘attach_node+0xb2
              genunix‘i_ndi_config_node+0x97
              genunix‘i_ddi_attachchild+0x4b
              genunix‘devi_attach_node+0x3d
              genunix‘devi_config_one+0x1d0
              genunix‘ndi_devi_config_one+0xb0
              devfs‘dv_find+0x125
              devfs‘devfs_lookup+0x40
              genunix‘fop_lookup+0x21
              genunix‘lookuppnvp+0x236
              genunix‘lookuppnat+0xe7
              genunix‘lookupnameat+0x87
              genunix‘cstatat_getvp+0x134

            value  ------------- Distribution ------------- count
             2048 |                                         0
             4096 |@@@@@@@@@@@@@@@@@@@@@@                   4105
             8192 |@@@@                                     783
            16384 |@@@@@@@@@@@@@@@                          2793
            32768 |                                         16
            65536 |                                         0


              kb8042‘kb8042_wait_poweron+0x29
              kb8042‘kb8042_init+0x22
              kb8042‘kb8042_attach+0xd6
              genunix‘devi_attach+0x75
              genunix‘attach_node+0xb2
              genunix‘i_ndi_config_node+0x97
              genunix‘i_ddi_attachchild+0x4b
              genunix‘devi_attach_node+0x3d
              genunix‘devi_config_one+0x1d0
              genunix‘ndi_devi_config_one+0xb0
              genunix‘resolve_pathname+0xa5
              genunix‘ddi_pathname_to_dev_t+0x16
              consconfig_dacf‘consconfig_load_drivers+0x14
              consconfig_dacf‘dynamic_console_config+0x6c
              consconfig‘consconfig+0x8
              unix‘stubs_common_code+0x3b

            value  ------------- Distribution ------------- count
           262144 |                                         0
           524288 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  221
          1048576 |@@@@                                     29
          2097152 |                                         0
```

```
              usba`hubd_enable_all_port_power+0xed
              usba`hubd_check_ports+0x8e
              usba`usba_hubdi_attach+0x275
              usba`usba_hubdi_bind_root_hub+0x168
              uhci`uhci_attach+0x191
              genunix`devi_attach+0x75
              genunix`attach_node+0xb2
              genunix`i_ndi_config_node+0x97
              genunix`i_ddi_attachchild+0x4b
              genunix`i_ddi_attach_node_hierarchy+0x49
              genunix`attach_driver_nodes+0x49
              genunix`ddi_hold_installed_driver+0xe3
              genunix`attach_drivers+0x28

           value  ------------- Distribution ------------- count
        33554432 |                                         0
        67108864 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
       134217728 |                                         0
```

Tail-Call-Optimierung

Wenn eine Funktion mit dem Aufruf einer anderen Funktion endet, kann der Compiler eine *Tail-Call-Optimierung* durchführen, die darin besteht, dass die aufgerufene Funktion den Stack-Frame des Aufrufers wieder verwendet. Dieses Verfahren findet in der SPARC-Architektur sehr häufige Anwendung, wo der Compiler das Registerfenster des Aufrufers in der aufgerufenen Funktion wieder verwendet, um den Druck auf die Registerfenster zu minimieren.

Diese Optimierung bewirkt, dass der `return`-Prüfpunkt der aufrufenden Funktion *vor* dem `entry`-Prüfpunkt der aufgerufenen Funktion ausgelöst wird. Diese Reihenfolge kann durchaus Verwirrung schaffen. Wenn Sie beispielsweise alle aus einer bestimmten Funktion aufgerufenen Funktionen und alle von ihr aufgerufenen Funktionen aufzeichnen möchten, könnten Sie das folgende Skript verwenden:

```
fbt::foo:entry
{
    self->traceme = 1;
}

fbt:::entry
/self->traceme/
{
    printf("called %s", probefunc);
}
```

```
fbt::foo:return
/self->traceme/
{
    self->traceme = 0;
}
```

Wenn jedoch `foo()` mit einem optimierten Endaufruf (Tail-Call) endet, wird die aus der Endposition rekursiv aufgerufene Funktion einschließlich aller Funktionen, die sie aufruft, nicht erfasst. Der Kernel kann nicht nach Bedarf dynamisch deoptimiert werden, und DTrace soll hier keine falschen Tatsachen über die Codestruktur vortäuschen. Deshalb sollte Ihnen bewusst sein, wann die Tail-Call-Optimierung möglicherweise angewendet wird.

Bei Quellcode in der Art des folgenden Beispiels ist die Verwendung der Tail-Call-Optimierung wahrscheinlich:

```
    return (bar());
```

Auch in Quellcode wie diesem:

```
    (void) bar();
    return;
```

Umgekehrt kann der Aufruf von *bar* in Funktionsquellcode, der wie das folgende Beispiel endet, `nicht()` optimiert werden, da der Aufruf von `bar()` kein Endaufruf ist:

```
    bar();
    return (rval);
```

Um festzustellen, ob ein Aufruf einer Tail-Call-Optimierung unterzogen wurde, gehen Sie wie folgt vor:

- Verfolgen Sie, während Sie DTrace ausführen, `arg0` des betreffenden `return`-Prüfpunkts. `arg0` enthält den Versatz der zurückkehrenden Anweisung in der Funktion.
- `mdb(1)` Wenn der verfolgte Versatz einen Aufruf einer anderen Funktion anstatt einer von der Funktion zurückkehrenden Anweisung enthält, wurde der Aufruf Tail-Call-optimiert.

Aufgrund der Befehlssatzarchitektur ist die Tail-Call-Optimierung auf SPARC-Systemen wesentlich verbreiteter als auf x86-Systemen. In diesem Beispiel wird `mdb` eingesetzt, um die Tail-Call-Optimierung in der Kernelfunktion `dup()` zu entdecken:

```
# dtrace -q -n fbt::dup:return'{printf("%s+0x%x", probefunc, arg0);}'
```

Führen Sie, während dieser Befehl läuft, ein Programm aus, das ein `dup(2)` durchführt, zum Beispiel einen `bash`-Prozess. Der obige Befehl sollte eine Ausgabe wie in folgendem Beispiel liefern:

```
dup+0x10
^C
```

Untersuchen Sie die Funktion nun mit `mdb`:

```
# echo "dup::dis" | mdb -k
dup:                            sra       %o0, 0, %o0
dup+4:                          mov       %o7, %g1
dup+8:                          clr       %o2
dup+0xc:                        clr       %o1
dup+0x10:                       call      -0x1278       <fcntl>
dup+0x14:                       mov       %g1, %o7
```

Die Ausgabe zeigt, dass `dup+0x10` ein Aufruf der Funktion `fcntl()` und keine `ret`-Anweisung ist. Der Aufruf von `fcntl()` ist also ein Beispiel für eine Tail-Call-Optimierung.

Assemblerfunktionen

Vielleicht fallen Ihnen Funktionen auf, die zwar einzutreten, aber nie zurückzukehren scheinen oder umgekehrt. Bei diesen seltenen Funktionen handelt es sich um handcodierte Assemblerroutinen, die sich zur Mitte anderer handcodierter Assemblerfunktionen verzweigen. Sie sollten die Analyse nicht behindern: Die Funktion am Verzweigungsziel muss trotzdem zum Aufrufer der Funktion zurückkehren, von der die Verzweigung ausgeht. Das bedeutet, dass Sie bei Aktivierung aller FBT-Prüfpunkte den Eintritt in eine Funktion und die Rückkehr von einer anderen Funktion in derselben Stacktiefe sehen sollten.

Beschränkungen des Befehlssatzes

Einige Funktionen können nicht mit FBT instrumentiert werden. Die genaue Natur nicht instrumentierbarer Funktionen ist kennzeichnend für die jeweilige Befehlssatzarchitektur.

Beschränkungen bei x86-Systemen

Funktionen, die auf x86-Systemen keinen Stack-Frame erzeugen, können nicht mit FBT instrumentiert werden. Da der Registersatz für x86 außerordentlich klein ist, müssen die meisten Funktionen Daten auf den Stapel legen und deshalb einen Stack-Frame erzeugen. Einige x86-Funktionen erzeugen jedoch keinen Stack-Frame und können folglich nicht instrumentiert werden. Die tatsächlichen Zahlen variieren von System zu System. Es lässt sich aber feststellen, dass in der Regel weniger als fünf Prozent der Funktionen auf der x86-Plattform nicht instrumentierbar sind.

Beschränkungen bei SPARC-Systemen

In Assemblersprache handcodierte Blattroutinen auf SPARC-Systemen können mit FBT nicht instrumentiert werden. Der Großteil des Kernels ist in C geschrieben, und alle in C geschriebenen Funktionen sind durch FBT instrumentierbar.

Interaktion mit Haltepunkten

Die Funktionsweise von FBT beruht auf der dynamischen Änderung von Kerneltext. Da auch Kernel-Haltepunkte auf diese Weise funktionieren, verweigert FBT die Bereitstellung eines Prüfpunkts für eine Funktion, wenn *vor* dem Laden von DTrace an der Eintritts- oder Rückkehrposition ein Kernel-Haltepunkt gesetzt wurde - selbst dann, wenn der Kernel-Haltepunkt anschließend entfernt wird. Wird der Kernel-Haltepunkt *nach* dem Laden von DTrace gesetzt, beziehen sich sowohl der Haltepunkt als auch der DTrace-Prüfpunkt auf dieselbe Stelle im Text. In dieser Situation spricht der Haltepunkt zuerst an und der Prüfpunkt wird anschließend ausgelöst, wenn der Debugger den Kernel wieder aufnimmt. Es empfiehlt sich, Kernel-Haltepunkte nicht gleichzeitig mit DTrace zu benutzen. Greifen Sie bei Bedarf stattdessen auf die DTrace-Aktion `breakpoint()` zurück.

Laden von Modulen

Der Solaris-Kernel kann Kernelmodule dynamisch laden und entladen. Wenn FBT geladen ist und ein Modul dynamisch geladen wird, liefert FBT automatisch neue Prüfpunkte für das neue Modul. Liegen für ein geladenes Modul *nicht aktivierte* FBT-Prüfpunkte vor, kann das Modul entladen (aus dem Speicher entfernt) werden; die zugehörigen Prüfpunkte werden beim Entladen des Moduls zerstört. Wenn für ein geladenes Modul *aktivierte* FBT-Prüfpunkte vorliegen, wird das Modul als belegt betrachtet und kann nicht aus dem Speicher entfernt werden.

Stabilität

Der Provider FBT beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Argumente	Private	Private	ISA

Wenn FBT die Kernel-Implementierung offen legt, ist nichts daran stabil („Stable“). Die Stabilität von Modul- und Funktionsnamen sowie der Daten ist explizit „Private“. Die Datenstabilität für Provider und Name ist „Evolving“, alle anderen Datenstabilitäten sind „Private“. Sie sind Artefakte der aktuellen Implementierung. Die Abhängigkeitsklasse (dependency class) für FBT lautet ISA: FBT ist zwar auf allen aktuellen Befehlssatzarchitekturen verfügbar, es besteht aber keine Garantie, dass FBT auch in zukünftigen Befehlssatzarchitekturen zur Verfügung stehen wird.

◆ ◆ ◆ KAPITEL 21

Der Provider `syscall`

Der Provider `syscall` stellt einen Prüfpunkt am Eintritt in und an der Rückkehr von jedem Systemaufruf im System zur Verfügung. Da Systemaufrufe die primäre Schnittstelle zwischen Anwendungen auf Benutzerebene und dem Betriebssystemkernel darstellen, kann Ihnen der Provider `syscall` einen erstaunlich tiefen Einblick in das Verhalten von Anwendungen in Bezug auf das System eröffnen.

Prüfpunkte

`syscall` stellt pro Systemaufruf ein Prüfpunktpaar bereit: einen `entry`-Prüfpunkt, der vor Eintritt in den Systemaufruf ausgelöst wird, und einen `return`-Prüfpunkt, der nach Abschluss des Systemaufrufs, aber noch vor der Rückgabe der Kontrolle an die Benutzerebene ausgelöst wird. Für alle `syscall`-Prüfpunkte wird der Funktionsname auf den Namen des instrumentierten Systemaufrufs gesetzt. Der Modulname ist nicht definiert.

Die Namen der Systemaufrufe, wie sie vom Provider `syscall` bereitgestellt werden, finden Sie in der Datei `/etc/name_to_sysnum`. Die Systemaufrufnamen, die `syscall` liefert, stimmen häufig mit den Namen in Section 2 der Manpages überein. Doch einige der Prüfpunkte des Providers `syscall` entsprechen keinem der dokumentierten Systemaufrufe direkt. Dieser Abschnitt klärt über die Gründe für diese Diskrepanz auf.

Systemaufruf-Anachronismen

In einigen Fällen spiegelt der Systemaufrufname, wie er vom Provider `syscall` bereitgestellt wird, eigentlich ein veraltetes Implementierungsdetail wider. So lautet beispielsweise der Name von `exit(2)` in `/etc/name_to_sysnum` aus Gründen, die weit in die UNIX™-Geschichte zurückreichen, `rexit`. Ebenso heißt `time(2)` `gtime`, und sowohl `execle(2)` als auch `execve(2)` heißen `exece`.

Subkodierte Systemaufrufe

Einige der in Section 2 vorgestellten Systemaufrufe sind als Unteroperationen eines nicht dokumentierten Systemaufrufs implementiert. So sind beispielsweise die Systemaufrufe für System-V-Semaphoren (`semctl(2)`, `semget(2)`, `semids(2)`, `semop(2)` und `semtimedop(2)`) als Unteroperationen eines einzigen Systemaufrufs, nämlich `semsys`, implementiert. Der Systemaufruf `semsys` übernimmt als erstes Argument einen implementierungsspezifischen *Subcode*, der den erforderlichen Systemaufruf angibt: `SEMCTL`, `SEMGET`, `SEMIDS`, `SEMOP` bzw. `SEMTIMEDOP`. Durch die Überladung eines einzelnen Systemaufrufs zur Implementierung mehrerer Systemaufrufe steht für System-V-Semaphoren nur ein einziges Paar syscall-Prüfpunkte zur Verfügung: `syscall::semsys:entry` und `syscall::semsys:return`.

Systemaufrufe für große Dateien

Ein 32-Bit-Programm, das *große Dateien* von über vier GB unterstützt, muss 64-Bit-Dateiversätze verarbeiten können. Da für eine große Datei auch immer ein großer Versatz verwendet werden muss, werden große Dateien über einen parallelen Satz Systemschnittstellen manipuliert. Dies ist in `lf64(5)` beschrieben. Diese Schnittstellen sind in `lf64` dokumentiert. Es liegen jedoch keine einzelnen Manpages für sie vor. Jede dieser Schnittstellen für Systemaufrufe im Zusammenhang mit großen Dateien erscheint, wie aus Tabelle 21–1 hervorgeht, als ein eigener `syscall`-Prüfpunkt.

TABELLE 21–1 `syscall`-Prüfpunkte für große Dateien

`syscall`-Prüfpunkt für große Dateien	Systemaufruf
`creat64`	`creat(2)`
`fstat64`	`fstat(2)`
`fstatvfs64`	`fstatvfs(2)`
`getdents64`	`getdents(2)`
`getrlimit64`	`getrlimit(2)`
`lstat64`	`lstat(2)`
`mmap64`	`mmap(2)`
`open64`	`open(2)`
`pread64`	`pread(2)`
`pwrite64`	`pwrite(2)`
`setrlimit64`	`setrlimit(2)`

TABELLE 21–1 `syscall`-Prüfpunkte für große Dateien *(Fortsetzung)*

`syscall`-Prüfpunkt für große Dateien	Systemaufruf
`stat64`	`stat(2)`
`statvfs64`	`statvfs(2)`

Private Systemaufrufe

Bei einigen Systemaufrufen handelt es sich um private Implementierungsdetails von Solaris-Subsystemen, die sich über die Grenze zwischen Benutzer- und Kernelebene erstrecken. Diese Systemaufrufe besitzen deshalb keine Manpages in Abschnitt 2. Zu den Systemaufrufen in dieser Kategorie gehören beispielsweise der Systemaufruf `signotify`, der im Rahmen der Implementierung von POSIX.4-Nachrichtenwarteschlangen verwendet wird, und der Systemaufruf `utssys`, der zur Implementierung von `fuser(1M)` dient.

Argumente

Bei `entry`-Prüfpunkten stellen die Argumente (`arg0` .. `arg`*n*`)` die `Argumente für den Systemaufruf dar.` Bei `return`-Prüfpunkten enthalten sowohl `arg0` als auch `arg1` den Rückgabewert. Ein Wert ungleich Null in der D-Variable `errno` weist auf einen Fehlschlag des Systemaufrufs hin.

Stabilität

Der Provider `syscall` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	Common
Modul	Private	Private	Unknown
Funktion	Unstable	Unstable	ISA
Name	Evolving	Evolving	Common
Argumente	Unstable	Unstable	ISA

◆ ◆ ◆ KAPITEL 22

Der Provider `sdt`

Der SDT-Provider (Statically Defined Tracing) erzeugt Prüfpunkte an Positionen, die vom Programmierer formal bestimmt werden. Der SDT-Mechanismus ermöglicht es Programmierern, den DTrace-Benutzern bewusst bestimmte Positionen anzubieten und ihnen mit dem Prüfpunktnamen semantische Kenntnisse über die einzelnen Positionen zu vermitteln. Im Solaris-Kernel sind eine Handvoll SDT-Prüfpunkte definiert, die voraussichtlich weiter zunehmen werden. DTrace stellt darüber hinaus·für Entwickler von Benutzeranwendungen einen Mechanismus zur Definition statischer Prüfpunkte bereit (siehe Kapitel 34, „Statisch definierte Ablaufverfolgung für Benutzeranwendungen“).

Prüfpunkte

Die vom Solaris-Kernel definierten SDT-Prüfpunkte sind in Tabelle 22–1 aufgeführt. Die Namensstabilität und Datenstabilität dieser Prüfpunkte ist „Private“. Ihre Beschreibung spiegelt damit die Kernel-Implementierung wider und sollte nicht als Schnittstellenbindung gedeutet werden. Weitere Informationen zum DTrace-Stabilitätsmechanismus finden Sie unter „Stabilität“ auf Seite 245.

TABELLE 22–1 SDT-Prüfpunkte

Prüfpunktname	Beschreibung	`arg0`
`callout-start`	Prüfpunkt, der unmittelbar vor der Ausführung eines Callouts ausgelöst wird (siehe `<sys/callo.h>`). Die Zeitüberschreitungen des Typs Callout werden durch regelmäßigen Systemtakt ausgeführt und stellen die Implementierung von `timeout(9F)` dar.	Zeiger auf `callout_t` (siehe `<sys/callo.h>`) für den auszuführenden Callout.

TABELLE 22–1 SDT-Prüfpunkte *(Fortsetzung)*

Prüfpunktname	Beschreibung	arg0
`callout-end`	Prüfpunkt, der unmittelbar nach der Ausführung eines Callouts ausgelöst wird (siehe `<sys/callo.h>`).	Zeiger auf `callout_t` (siehe `<sys/callo.h>`) für den gerade ausgeführten Callout.
`interrupt-start`	Prüfpunkt, der unmittelbar vor dem Aufruf in eine Interrupt-Behandlungsroutine eines Geräts ausgelöst wird.	Zeiger auf die Struktur `dev_info` (siehe `<sys/ddi_impldefs.h>`) für das unterbrechende Gerät.
`interrupt-complete`	Prüfpunkt, der unmittelbar nach der Rückkehr von der Interrupt-Behandlungsroutine eines Geräts ausgelöst wird.	Zeiger auf die Struktur `dev_info` (siehe `<sys/ddi_impldefs.h>`) für das unterbrechende Gerät.

Beispiele

Das folgende Beispiel ist ein Skript zur Beobachtung des Callout-Verhaltens auf Sekundenbasis:

```
#pragma D option quiet

sdt:::callout-start
{
    @callouts[((callout_t *)arg0)->c_func] = count();
}

tick-1sec
{
    printa("%40a %10@d\n", @callouts);
    clear(@callouts);
}
```

Dieses Beispielskript deckt die häufigen Benutzer von `timeout(9F)` im System auf, wie die folgende Ausgabe zeigt:

```
# dtrace -s ./callout.d
                                     FUNC      COUNT
                              TS‘ts_update          1
              uhci‘uhci_cmd_timeout_hdlr          3
                          genunix‘setrun          5
                    genunix‘schedpaging           5
                        ata‘ghd_timeout          10
 uhci‘uhci_handle_root_hub_status_change         309

                                     FUNC      COUNT
              ip‘tcp_time_wait_collector          1
```

```
                              TS`ts_update          1
                 uhci`uhci_cmd_timeout_hdlr          3
                        genunix`schedpaging          4
                             genunix`setrun          8
                            ata`ghd_timeout         10
 uhci`uhci_handle_root_hub_status_change        300

                                      FUNC      COUNT
                 ip`tcp_time_wait_collector          0
                           iprb`mii_portmon          1
                              TS`ts_update          1
                 uhci`uhci_cmd_timeout_hdlr          3
                        genunix`schedpaging          4
                             genunix`setrun          7
                            ata`ghd_timeout         10
 uhci`uhci_handle_root_hub_status_change        300
```

Die Schnittstelle timeout(9F) erzeugt nur einen einzelnen Timer-Ablauf. Verbraucher von `timeout()`, die eine intervallbasierte Timer-Funktion benötigen, installieren in der Regel das Timeout ihrer `timeout()`-Behandlungsroutine neu. Das folgende Beispiel demonstriert dieses Verhalten:

```
#pragma D option quiet

sdt:::callout-start
{
    self->callout = ((callout_t *)arg0)->c_func;
}

fbt::timeout:entry
/self->callout && arg2 <= 100/
{
    /*
     * In this case, we are most interested in interval timeout(9F)s that
     * are short.  We therefore do a linear quantization from 0 ticks to
     * 100 ticks.  The system clock's frequency — set by the variable
     * "hz" — defaults to 100, so 100 system clock ticks is one second.
     */
    @callout[self->callout] = lquantize(arg2, 0, 100);
}

sdt:::callout-end
{
    self->callout = NULL;
}

END
{
```

```
    printa("%a\n%@d\n\n", @callout);
}
```

Wenn Sie dieses Skript ausführen, einige Sekunden warten und dann Strg-C eingeben, erhalten Sie eine ähnliche Ausgabe wie diese:

```
# dtrace -s ./interval.d
^C
genunix`schedpaging

           value  ------------- Distribution ------------- count
              24 |                                         0
              25 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 20
              26 |                                         0


ata`ghd_timeout

           value  ------------- Distribution ------------- count
               9 |                                         0
              10 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 51
              11 |                                         0


uhci`uhci_handle_root_hub_status_change

           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1515
               2 |                                         0
```

Die Ausgabe zeigt, dass `uhci_handle_root_hub_status_change()` im Treiber uhci(7D) den Timer mit dem kürzesten Intervall auf dem System darstellt: Er wird mit jedem Systemuhr-Tick aufgerufen.

Mit dem Prüfpunkt `interrupt-start` lässt sich die Interrupt-Aktivität nachvollziehen. Das folgende Beispiel zeigt, wie die für die Ausführung einer Interrupt-Behandlungsroutine benötigte Zeit je Treibername quantisiert werden kann:

```
interrupt-start
{
    self->ts = vtimestamp;
}

interrupt-complete
/self->ts/
{
    this->devi = (struct dev_info *)arg0;
```

```
    @[stringof(‘devnamesp[this->devi->devi_major].dn_name),
        this->devi->devi_instance] = quantize(vtimestamp - self->ts);
}
```

Die Ausführung dieses Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./intr.d
dtrace: script ’./intr.d’ matched 2 probes
^C
 isp                                                     0
           value  ------------- Distribution ------------- count
            8192 |                                         0
           16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
           32768 |                                         0

  pcf8584                                                 0
           value  ------------- Distribution ------------- count
              64 |                                         0
             128 |                                         2
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@         157
             512 |@@@@@@                                   31
            1024 |                                         3
            2048 |                                         0

  pcf8584                                                 1
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 154
            8192 |@@@@@@@                                  37
           16384 |                                         2
           32768 |                                         0

  qlc                                                     0
           value  ------------- Distribution ------------- count
           16384 |                                         0
           32768 |@@                                       9
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 126
          131072 |@                                        5
          262144 |                                         2
          524288 |                                         0

  hme                                                     0
           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |                                         6
            4096 |                                         2
            8192 |@@@@                                     89
           16384 |@@@@@@@@@@@@@@@                          262
```

```
          32768 |@                                        37
          65536 |@@@@@@@                                  139
         131072 |@@@@@@@@                                 161
         262144 |@@@                                      73
         524288 |                                         4
        1048576 |                                         0
        2097152 |                                         1
        4194304 |                                         0

ohci                                                      0
          value  ------------- Distribution ------------- count
           8192 |                                         0
          16384 |                                         3
          32768 |                                         1
          65536 |@@@                                      143
         131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     1368
         262144 |                                         0
```

Erstellen von SDT-Prüfpunkten

Entwickler von Gerätetreibern interessiert es wahrscheinlich, wie man eigene SDT-Prüfpunkte im Solaris-Treiber erstellen kann. Die ausgeschaltete Prüfaktivität von SDT ist im Wesentlichen der Preis für verschiedene NOP-Maschinenanweisungen. Sie können Ihren Gerätetreibern deshalb ganz nach Bedarf neue SDT-Prüfpunkte hinzufügen. Sofern sie die Leistung nicht negativ beeinflussen, besteht kein Grund, die Prüfpunkte nicht im Auslieferungszustand des Codes beizubehalten.

Deklarieren von Prüfpunkten

SDT-Prüfpunkte werden mit den Makros `DTRACE_PROBE`, `DTRACE_PROBE1`, `DTRACE_PROBE2`, `DTRACE_PROBE3` und `DTRACE_PROBE4` aus `<sys/sdt.h>` deklariert. Der Modulname und Funktionsname eines auf SDT basierenden Prüfpunkts entspricht dem Kernelmodul und der Funktion des Prüfpunkts. Der Name des Prüfpunkts ist von dem Namen in der Makro `DTRACE_PROBE`*n* `abhängig.` Wenn der Name keine zwei aufeinander folgenden Unterstriche (`__`) enthält, ist der Name des Prüfpunkts mit der Schreibweise in der Makro identisch. Enthält er jedoch zwei aufeinander folgende Unterstriche, werden diese im Prüfpunktnamen in einen Gedankenstrich (`-`) umgewandelt. Wenn beispielsweise eine `DTRACE_PROBE`-Makro `transaction__start` angibt, heißt der SDT-Prüfpunkt `transaction-start`. Durch diese Ersetzung kann C-Code Makronamen bereitstellen, die keine gültigen C-Bezeichner sind, ohne eine Zeichenkette anzugeben.

DTrace nimmt den Kernelmodul- und den Funktionsnamen bereits als Bestandteil des Tupels auf, das einen Prüfpunkt bezeichnet. Sie brauchen diese Angaben also nicht mehr in den Prüfpunktnamen einzufügen, um Namensraumkollisionen zu vermeiden. Sie können den

Befehl `dtrace -l -P sdt -m` *Modul* auf das Treiber*modul* anwenden, um die installierten Prüfpunkte sowie die für DTrace-Benutzer sichtbaren vollständigen Namen aufzulisten.

Prüfpunktargumente

Die Argumente der einzelnen SDT-Prüfpunkte sind die im entsprechenden `DTRACE_PROBE`*n*-Makroverweis angegebenen Argumente. Die Anzahl der Argumente hängt davon ab, welche Makro zum Erstellen des Prüfpunkts verwendet wurde: `DTRACE_PROBE1` gibt ein Argument an, `DTRACE_PROBE2` zwei usw. Beim Deklarieren der SDT-Prüfpunkte können Sie ihre ausgeschaltete Prüftätigkeit auf ein Minimum herabsetzen, indem Sie keine Zeiger dereferenzieren und keine Daten aus globalen Variablen in den Prüfpunktargumenten laden. Sowohl die Zeigerdereferenzierung als auch das Laden aus globalen Variablen kann über D-Aktionen, die Prüfpunkte aktivieren, sicher vorgenommen werden. Auf diese Weise können DTrace-Benutzer diese Aktionen nur dann anfordern, wenn sie benötigt werden.

Stabilität

Der Provider SDT beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Private	Private	ISA
Argumente	Private	Private	ISA

◆ ◆ ◆ KAPITEL 23

Der Provider sysinfo

Der Provider sysinfo stellt Prüfpunkte für Kernelstatistiken zur Verfügung, die mit der Bezeichnung sys klassifiziert sind. Da auf diesen statistischen Daten Dienstprogramme zur Systemüberwachung wie etwa mpstat(1M) beruhen, ermöglicht der Provider sysinfo eine schnelle Untersuchung beobachteten Fehlverhaltens.

Prüfpunkte

Der Provider sysinfo stellt Prüfpunkte für die Felder unter der Bezeichnung sys in der Kernelstatistik bereit: Ein Prüfpunkt von sysinfo wird ausgelöst, unmittelbar bevor der entsprechende sys-Wert erhöht wird. Das folgende Beispiel zeigt, wie mithilfe des Befehls kstat(1M) sowohl die Namen als auch die aktuellen Werte der mit sys bezeichneten Kernelstatistiken angezeigt werden können.

```
$ kstat -n sys
module: cpu                             instance: 0
name:   sys                             class:    misc
    bawrite                         123
    bread                           2899
    bwrite                          17995
...
```

Die sysinfo-Prüfpunkte sind in Tabelle 23–1 beschrieben.

TABELLE 23–1 sysinfo-Prüfpunkte

bawrite	Prüfpunkt, der immer kurz vor der asynchronen Ausgabe eines Puffers an ein Gerät ausgelöst wird.

TABELLE 23–1 `sysinfo`-Prüfpunkte *(Fortsetzung)*

`bread`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Gerät einen Puffer physisch ausliest. `bread` wird ausgelöst, *nachdem* der Puffer vom Gerät angefordert wurde, aber noch *vor* der Blockierung in Erwartung der Durchführung.
`bwrite`	Prüfpunkt, der immer kurz vor der synchronen *oder* asynchronen Ausgabe eines Puffers an ein Gerät ausgelöst wird.
`idlethread`	Prüfpunkt der ausgelöst wird, wenn eine CPU die Leerlaufschleife betritt.
`intrblk`	Prüfpunkt der ausgelöst wird, wenn ein Interrupt-Thread blockiert wird.
`inv_swtch`	Prüfpunkt, der ausgelöst wird, wenn ein laufender Thread gezwungen ist, die CPU aufzugeben.
`lread`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Gerät einen Puffer logisch ausliest.
`lwrite`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Puffer logisch auf ein Gerät geschrieben wird.
`modload`	Prüfpunkt der bei jedem Laden eines Kernelmoduls ausgelöst wird.
`modunload`	Prüfpunkt der bei jedem Entfernen eines Kernelmoduls aus dem Speicher ausgelöst wird.
`msg`	Prüfpunkt, der bei jedem `msgsnd(2)`- oder `msgrcv(2)`-Systemaufruf, aber noch vor der Durchführung der Message-Queue-Operationen ausgelöst wird.
`mutex_adenters`	Prüfpunkt, der bei jedem Versuch ausgelöst wird, eine belegte adaptive Sperre zu erlangen. Wenn dieser Prüfpunkt ausgelöst wird, wird auch einer der Prüfpunkte `adaptive-block` oder `adaptive-spin` des Providers `lockstat` ausgelöst. Ausführliche Informationen finden Sie in Kapitel 18, „Der Provider `lockstat`“.
`namei`	Prüfpunkt, der ausgelöst wird, wenn ein Name im Dateisystem gesucht (lookup) wird.
`nthreads`	Prüfpunkt, der bei jeder Erzeugung eines Threads ausgelöst wird.
`phread`	Prüfpunkt, der ausgelöst wird, wenn ein Raw-E/A-Lesezugriff ansteht.
`phwrite`	Prüfpunkt, der ausgelöst wird, wenn ein Raw-E/A-Schreibzugriff ansteht.
`procovf`	Prüfpunkt, der ausgelöst wird, wenn ein neuer Prozess nicht erzeugt werden kann, da dem System keine Prozesstabelleneinträge mehr zur Verfügung stehen.
`pswitch`	Prüfpunkt, der ausgelöst wird, wenn eine CPU von der Ausführung des einen zur Ausführung eines anderen Threads wechselt.
`readch`	Prüfpunkt, der nach jedem erfolgreichen Lesezugriff ausgelöst wird, aber noch bevor die Kontrolle wieder an den den Lesezugriff durchführenden Thread übergeben wird. Ein Lesezugriff kann über die Systemaufrufe `read(2)`, `readv(2)` oder `pread(2)` erfolgen. `arg0` enthält die Anzahl der Byte, die erfolgreich gelesen wurden.

TABELLE 23–1 `sysinfo`-Prüfpunkte *(Fortsetzung)*

`rw_rdfails`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Versuch stattfindet, eine Lesesperre auf eine Leser/Schreiber-Sperre zu legen, wenn die Sperre entweder von einem Schreiber belegt ist oder gefordert wird. Wenn dieser Prüfpunkt ausgelöst wird, wird auch der Prüfpunkt `rw-block` des Providers `lockstat` ausgelöst. Ausführliche Informationen finden Sie in Kapitel 18, „Der Provider `lockstat`“.
`rw_wrfails`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Versuch stattfindet, eine Schreibsperre auf eine Leser/Schreiber-Sperre zu legen, wenn die Sperre entweder von Lesern oder einem anderen Schreiber belegt ist. Wenn dieser Prüfpunkt ausgelöst wird, wird auch der Prüfpunkt `rw-block` des Providers `lockstat` ausgelöst. Ausführliche Informationen finden Sie in Kapitel 18, „Der Provider `lockstat`“.
`sema`	Prüfpunkt, der bei jedem `semop(2)`-Systemaufruf, aber noch vor der Durchführung etwaiger Semaphor-Operationen ausgelöst wird.
`sysexec`	Prüfpunkt, der bei jedem `exec(2)`-Systemaufruf ausgelöst wird.
`sysfork`	Prüfpunkt, der bei jedem `fork(2)`-Systemaufruf ausgelöst wird.
`sysread`	Prüfpunkt, der bei jedem Systemaufruf von `read(2)`, `readv(2)` oder `pread(2)` ausgelöst wird.
`sysvfork`	Prüfpunkt, der bei jedem `vfork(2)`-Systemaufruf ausgelöst wird.
`syswrite`	Prüfpunkt, der bei jedem Systemaufruf von `write(2)`, `writev(2)` oder `pwrite(2)` ausgelöst wird.
`trap`	Prüfpunkt, der bei jedem Prozessor-Trap ausgelöst wird. Beachten Sie, dass einige Prozessoren, insbesondere Varianten von UltraSPARC, bestimmte leichtgewichtige Traps über einen Mechanismus behandeln, der diesen Prüfpunkt nicht auslöst.
`ufsdirblk`	Prüfpunkt, der ausgelöst wird, wenn ein Verzeichnisblock aus dem UFS-Dateisystem ausgelesen wird. Ausführliche Informationen zu UFS finden Sie unter `ufs(7FS)`.
`ufsiget`	Prüfpunkt, der bei jedem Abruf eines Inodes aufgerufen wird. Ausführliche Informationen zu UFS finden Sie unter `ufs(7FS)`.
`ufsinopage`	Prüfpunkt, der ausgelöst wird, nachdem ein Incore-Inode *ohne* zugehörige Datenseiten zur Wiederverwendung freigegeben wurde. Ausführliche Informationen zu UFS finden Sie unter `ufs(7FS)`.
`ufsipage`	Prüfpunkt, der ausgelöst wird, nachdem ein Incore-Inode *mit* zugehörigen Datenseiten zur Wiederverwendung freigegeben wurde. Dieser Prüfpunkt wird ausgelöst, nachdem die zugehörigen Datenseiten auf die Festplatte entleert wurden. Ausführliche Informationen zu UFS finden Sie unter `ufs(7FS)`.
`writech`	Prüfpunkt, der nach jedem erfolgreichen Schreibzugriff ausgelöst wird, aber noch bevor die Kontrolle wieder an den den Schreibzugriff durchführenden Thread übergeben wird. Ein Schreibzugriff kann über die Systemaufrufe `write(2)`, `writev(2)` oder `pwrite(2)` erfolgen. `arg0` enthält die Anzahl der Byte, die erfolgreich geschrieben wurden.

TABELLE 23–1 `sysinfo`-Prüfpunkte *(Fortsetzung)*

`xcalls`	Prüfpunkt, der immer kurz vor einem gegenseitigen Aufruf ausgelöst wird. Ein gegenseitiger Aufruf ist der Mechanismus des Betriebssystems, über den eine CPU den sofortigen Einsatz einer anderen CPU fordert.

Argumente

Die Argumente für `sysinfo`-Prüfpunkte lauten:

`arg0`	Der Wert, um den die statistische Angabe zu erhöhen ist. Bei den meisten Prüfpunkten beträgt der Wert dieses Arguments stets 1, bei einigen kann das Argument andere Werte annehmen.
`arg1`	Ein Zeiger auf den aktuellen Wert der zu erhöhenden statistischen Angabe. Dieser Wert ist eine 64-Bit-Größe, die um den Wert in `arg0` erhöht wird. Durch Dereferenzierung dieses Zeigers können die Verbraucher die aktuelle Gesamtzahl der zum Prüfpunkt gehörigen statistischen Angabe ermitteln.
`arg2`	Ein Zeiger auf die Struktur `cpu_t`, die für die CPU steht, auf der die statistische Angabe erhöht werden soll. Diese Struktur ist in `<sys/cpuvar.h>` definiert, ist aber Bestandteil der Kernel-Implementierung und sollte als „Private“ betrachtet werden.

Der Wert von `arg0` beträgt bei den meisten `sysinfo`-Prüfpunkten 1. Die Prüfpunkte `readch` und `writech` setzen `arg0` jedoch auf die Anzahl der gelesenen bzw. geschriebenen Byte. Mit diesem Leistungsmerkmal lässt sich, wie das folgende Beispiel zeigt, die Größe von Lesezugriffen nach Namen der ausführbaren Datei ermitteln:

```
# dtrace -n readch'{@[execname] = quantize(arg0)}'
dtrace: description 'readch' matched 4 probes
^C
  xclock
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
              64 |                                         0

  acroread
           value  ------------- Distribution ------------- count
              16 |                                         0
              32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
              64 |                                         0

  FvwmAuto
           value  ------------- Distribution ------------- count
               2 |                                         0
```

```
             4 |@@@@@@@@@@@@@@                          13
             8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 21
            16 |@@@@@@                                  5
            32 |                                        0

xterm
         value  ------------- Distribution ------------- count
            16 |                                        0
            32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 19
            64 |@@@@@@@@@@                              7
           128 |@@@@@@                                  5
           256 |                                        0

fvwm2
         value  ------------- Distribution ------------- count
            -1 |                                        0
             0 |@@@@@@@@@@                              186
             1 |                                        0
             2 |                                        0
             4 |@@                                      51
             8 |                                        17
            16 |                                        0
            32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 503
            64 |                                        9
           128 |                                        0

Xsun
         value  ------------- Distribution ------------- count
            -1 |                                        0
             0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 269
             1 |                                        0
             2 |                                        0
             4 |                                        2
             8 |@                                       31
            16 |@@@@@@                                  128
            32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 171
            64 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 33
           128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 85
           256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 24
           512 |                                        8
          1024 |                                        21
          2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 26
          4096 |                                        21
          8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 94
         16384 |                                        0
```

Der Provider `sysinfo` setzt `arg2` als einen Zeiger auf `cpu_t`, eine interne Struktur der Kernel-Implementierung. Die `sysinfo`-Prüfpunkte werden auf der CPU ausgelöst, auf der die Statistik inkrementiert wird. Mit der Komponente `cpu_id` der Struktur `cpu_t` bestimmen Sie die relevante CPU.

Beispiel

Betrachten Sie die folgende Ausgabe von `mpstat(1M)`:

```
CPU minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl
 12   90  22 5760   422  299  435   26   71  116   11  1372    5  19  17  60
 13   46  18 4585   193  162  431   25   69  117   12  1039    3  17  14  66
 14   33  13 3186   405  381  397   21   58  105   10   770    2  17  11  70
 15   34  19 4769   109   78  417   23   57  115   13   962    3  14  14  69
 16   74  16 4421   437  406  448   29   77  111    8  1020    4  23  14  59
 17   51  15 4493   139  110  378   23   62  109    9   928    4  18  14  65
 18   41  14 4204   494  468  360   23   56  102    9   849    4  17  12  68
 19   37  14 4229   115   87  363   22   50  106   10   845    3  15  14  67
 20   78  17 5170   200  169  456   26   69  108    9  1119    5  21  25  49
 21   53  16 4817    78   51  394   22   56  106    9   978    4  17  22  57
 22   32  13 3474   486  463  347   22   48  106    9   769    3  17  17  63
 23   43  15 4572    59   34  361   21   46  102   10   947    4  15  22  59
```

Aus dieser Ausgabe könnte man schließen, dass das Feld `xcal`, insbesondere angesichts des relativen Leerlaufs des Systems, zu hoch ausfällt. `mpstat` bestimmt den Wert im Feld `xcal` über das Feld `xcalls` in der `sys`-Kernelstatistik. Diese Abweichung lässt sich deshalb problemlos durch Aktivierung des Prüfpunkts `xcalls` `sysinfo` untersuchen, wie das nächste Beispiel zeigt:

```
# dtrace -n xcalls'{@[execname] = count()}'
dtrace: description 'xcalls' matched 4 probes
^C
  dtterm                                                            1
  nsrd                                                              1
  in.mpathd                                                         2
  top                                                               3
  lockd                                                             4
  java_vm                                                          10
  ksh                                                              19
  iCald.pl6+RPATH                                                  28
  nwadmin                                                          30
  fsflush                                                          34
  nsrindexd                                                        45
  in.rlogind                                                       56
  in.routed                                                       100
  dtrace                                                          153
  rpc.rstatd                                                      246
```

```
  imapd                                                             377
  sched                                                             431
  nfsd                                                             1227
  find                                                             3767
```

Die Ausgabe zeigt, wo nach der Quelle der gegenseitigen Aufrufe zu suchen ist. Einige `find(1)`-Prozesse verursachen den Großteil der gegenseitigen Aufrufe. Das nächste D-Skript hilft, dem Problem weiter auf den Grund zu gehen:

```
syscall:::entry
/execname == "find"/
{
    self->syscall = probefunc;
    self->insys = 1;
}

sysinfo:::xcalls
/execname == "find"/
{
    @[self->insys ? self->syscall : "<none>"] = count();
}

syscall:::return
/self->insys/
{
    self->insys = 0;
    self->syscall = NULL;
}
```

In diesem Skript wird der Provider `syscall` dazu verwendet, gegenseitige Aufrufe aus `find` einem bestimmten Systemaufruf zuzuordnen. Einige gegenseitige Aufrufe, wie solche, die aus Seitenfehlern entstehen, gehen nicht von Systemaufrufen aus. In diesen Fällen gibt das Skript „<none>” aus. Die Ausführung dieses Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./find.d
 dtrace: script ’./find.d’ matched 444 probes
^C
  <none>                                                              2
  lstat64                                                          2433
  getdents64                                                      14873
```

Diese Ausgabe deutet darauf hin, dass die Mehrheit der von `find` bewirkten gegenseitigen Aufrufe ihrerseits durch `getdents(2)`-Systemaufrufe verursacht wurden. Für eine weitere Untersuchung wäre die gewünschte Richtung ausschlaggebend. Wenn Sie verstehen möchten, weshalb `find`-Prozesse `getdents`-Aufrufe tätigen, könnten Sie ein D-Skript schreiben, das `ustack()` aggregiert, wenn `find` einen gegenseitigen Aufruf verursacht. Wenn es Sie interessiert, weshalb `getdents`-Aufrufe gegenseitige Aufrufe verursachen, bietet sich ein D-Skript an, das `ustack()` aggregiert, wenn `find` einen gegenseitigen Aufruf verursacht.

Welche Richtung Sie auch immer einschlagen, hat Ihnen der Prüfpunkt `xcalls` die Möglichkeit gegeben, die Ursache der ungewöhnlichen Überwachungsausgabe aufzudecken.

Stabilität

Der Provider `sysinfo` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Argumente	Private	Private	ISA

◆ ◆ ◆ KAPITEL 24

Der Provider `vminfo`

Der Provider `vminfo` stellt Prüfpunkte für die `vm`-Kernelstatistik zur Verfügung. Da auf diesen statistischen Daten Dienstprogramme zur Systemüberwachung wie etwa `vmstat(1M)` beruhen, ermöglicht der Provider `vminfo` eine schnelle Untersuchung beobachteten Fehlverhaltens.

Prüfpunkte

Der Provider `vminfo` stellt Prüfpunkte für die Felder unter der Bezeichnung `vm` in der Kernelstatistik bereit: Ein Prüfpunkt von `vminfo` wird ausgelöst, unmittelbar bevor der entsprechende `vm`-Wert erhöht wird. Das folgende Beispiel zeigt, wie mithilfe des Befehls `kstat(1M)` sowohl die Namen als auch die aktuellen Werte der mit `vm` bezeichneten Kernelstatistiken angezeigt werden können:

```
$ kstat -n vm
module: cpu                             instance: 0
name:   vm                              class:    misc
        anonfree                        13
        anonpgin                        2620
        anonpgout                       13
        as_fault                        12528831
        cow_fault                       2278711
        crtime                          202.10625712
        dfree                           1328740
        execfree                        0
        execpgin                        5541
        ...
```

Die `vminfo`-Prüfpunkte sind in Tabelle 24–1 beschrieben.

TABELLE 24–1 `vminfo`-Prüfpunkte

`anonfree`	Prüfpunkt der ausgelöst wird, wenn eine unveränderte, anonyme Speicherseite im Rahmen einer Paging-Aktivität freigegeben wird. Bei anonymen Speicherseiten handelt es sich um Seiten, die keiner Datei zugeordnet sind. Speicher, die solche Seiten enthalten, sind der Heap-, der Stack- oder der Speicher, der durch explizite Zuordnung von `zero(7D)` erhalten wird.
`anonpgin`	Prüfpunkt, der ausgelöst wird, wenn eine anonyme Speicherseite aus einem Swap-Gerät eingelagert wird.
`anonpgout`	Prüfpunkt, der ausgelöst wird, wenn eine veränderte, anonyme Speicherseite an ein Swap-Gerät ausgelagert wird.
`as_fault`	Prüfpunkt, der ausgelöst wird, wenn ein Seitenfehler auftritt, bei dem es sich weder um einen Schutz- noch einen Copy-on-Write-Fehler handelt.
`cow_fault`	Prüfpunkt, der mit jedem Copy-on-Write-Fehler für eine Speicherseite ausgelöst wird. `arg0` enthält die Anzahl der als Folge des Copy-on-Write-Vorgangs erzeugten Speicherseiten.
`dfree`	Prüfpunkt, der ausgelöst wird, wenn eine Speicherseite als Folge einer Paging-Aktivität freigegeben wird. Auf jede Auslösung von `dfree` folgt immer genau eine Auslösung von `anonfree`, `execfree` oder `fsfree`.
`execfree`	Prüfpunkt, der ausgelöst wird, wenn eine unveränderte, ausführbare Speicherseite als Folge einer Paging-Aktivität freigegeben wird.
`execpgin`	Prüfpunkt, der ausgelöst wird, wenn eine ausführbare Speicherseite aus einem Zusatzspeicher eingelagert wird.
`execpgout`	Prüfpunkt, der ausgelöst wird, wenn eine veränderte, ausführbare Speicherseite an einen Zusatzspeicher ausgelagert wird. Das Paging ausführbarer Seiten erfolgt meistens in der Form `execfree`. `execpgout` kann nur ausgeführt werden, wenn eine ausführbare Seite im Speicher modifiziert wird, was auf den meisten Systemen äußerst ungewöhnlich ist.
`fsfree`	Prüfpunkt der ausgelöst wird, wenn eine unveränderte Dateisystem-Datenseite im Rahmen einer Paging-Aktivität freigegeben wird.
`fspgin`	Prüfpunkt, der ausgelöst wird, wenn eine Dateisystemseite aus einem Zusatzspeicher eingelagert wird.
`fspgout`	Prüfpunkt, der ausgelöst wird, wenn eine veränderte Dateisystemseite an einen Zusatzspeicher ausgelagert wird.
`kernel_asflt`	Prüfpunkt, der mit jedem Seitenfehler im Adressraum des Kernels ausgelöst wird. Jeder Auslösung von `kernel_asflt` geht unmittelbar eine Auslösung des Prüfpunkts `as_fault` voraus.
`maj_fault`	Prüfpunkt, der ausgelöst wird, wenn ein Seitenfehler auftritt, der E/A von einem Zusatzspeicher oder Swap-Gerät zur Folge hat. Jeder Auslösung von `maj_fault` geht unmittelbar eine Auslösung des Prüfpunkts `pgin` voraus.

TABELLE 24–1 `vminfo`-Prüfpunkte *(Fortsetzung)*

`pgfrec`	Prüfpunkt, der ausgelöst wird, wenn eine Seite aus der Liste der freien Seiten gefordert wird.
`pgin`	Prüfpunkt, der ausgelöst wird, wenn eine Seite aus dem Zusatzspeicher oder einem Swap-Gerät eingelagert wird. Dieser Prüfpunkt unterscheidet sich insofern von `maj_fault`, als `maj_fault` nur ausgelöst wird, wenn eine Seite als Folge eines Seitenfehlers eingelagert wird. `pgin` wird immer dann ausgelöst, wenn eine Seite eingelagert wird - egal aus welchem Grund.
`pgout`	Prüfpunkt, der ausgelöst wird, wenn eine Seite an den Zusatzspeicher oder ein Swap-Gerät ausgelagert wird.
`pgpgin`	Prüfpunkt, der ausgelöst wird, wenn eine Seite aus dem Zusatzspeicher oder einem Swap-Gerät eingelagert wird. Der einzige Unterschied zwischen `pgpgin` und `pgin` besteht darin, dass `pgpgin` die Anzahl der eingelagerten Seiten als `arg0` enthält. `pgin` enthält in `arg0` immer 1.
`pgpgout`	Prüfpunkt, der ausgelöst wird, wenn eine Seite an den Zusatzspeicher oder ein Swap-Gerät ausgelagert wird. Der einzige Unterschied zwischen `pgpgout` und `pgout` besteht darin, dass `pgpgout` die Anzahl der ausgelagerten Seiten als `arg0` enthält. (`pgout` enthält in `arg0` immer 1.)
`pgrec`	Prüfpunkt, der bei jeder Forderung einer Speicherseite ausgelöst wird.
`pgrrun`	Prüfpunkt, der immer dann ausgelöst wird, wenn der Pager eingeplant ist.
`pgswapin`	Prüfpunkt, der beim Einlagern von Seiten eines ausgelagerten Prozesses ausgelöst wird. `arg0` enthält die Anzahl der eingelagerten Seiten.
`pgswapout`	Prüfpunkt, der ausgelöst wird, wenn mit dem Auslagern eines Prozesses Seiten ausgelagert werden. `arg0` enthält die Anzahl der ausgelagerten Seiten.
`prot_fault`	Prüfpunkt, der ausgelöst wird, wenn ein Seitenfehler aufgrund einer Schutzverletzung auftritt.
`rev`	Prüfpunkt, der ausgelöst wird, wenn der Page-Dämon beginnt, alle Speicherseiten erneut zu durchlaufen.
`scan`	Prüfpunkt, der ausgelöst wird, wenn der Page-Dämon eine Speicherseite prüft.
`softlock`	Prüfpunkt, der ausgelöst wird, wenn ein Seitenfehler auftritt, weil diese Seite mit einer Softwaresperre belegt wurde.
`swapin`	Prüfpunkt, der ausgelöst wird, wenn ein ausgelagerter Prozess wieder eingelagert wird.
`swapout`	Prüfpunkt, der bei jeder Auslagerung eines Prozesses ausgelöst wird.
`zfod`	Prüfpunkt, der ausgelöst wird, wenn eine mit Nullen gefüllte Seite auf Anfrage erzeugt wird.

Argumente

`arg0`	Der Wert, um den die statistische Angabe zu erhöhen ist. Bei den meisten Prüfpunkten beträgt der Wert dieses Arguments stets 1, bei einigen kann das Argument andere Werte annehmen. Diese Prüfpunkte sind in Tabelle 24–1 aufgeführt.
`arg1`	Ein Zeiger auf den aktuellen Wert der zu erhöhenden statistischen Angabe. Dieser Wert ist eine 64-Bit-Größe, die um den Wert in `arg0` erhöht wird. Durch Dereferenzierung dieses Zeigers können die Verbraucher die aktuelle Gesamtzahl der zum Prüfpunkt gehörigen statistischen Angabe ermitteln.

Beispiel

Betrachten Sie die folgende Ausgabe von `vmstat(1M)`:

```
kthr      memory            page            disk          faults      cpu
 r b w   swap  free  re  mf pi po fr de sr cd s0 -- --   in   sy   cs us sy id
 0 1 0 1341844 836720 26 311 1644 0 0 0  0 216 0  0  0  797  817  697  9 10 81
 0 1 0 1341344 835300 238 934 1576 0 0 0 0 194 0  0  0  750 2795  791  7 14 79
 0 1 0 1340764 833668 24 165 1149 0 0 0  0 133 0  0  0  637  813  547  5  4 91
 0 1 0 1340420 833024 24 394 1002 0 0 0  0 130 0  0  0  621 2284  653 14  7 79
 0 1 0 1340068 831520 14 202 380 0 0  0  0 59  0  0  0  482 5688 1434 25  7 68
```

Die Spalte `pi` in der obigen Ausgabe gibt die Anzahl der eingelagerten Seiten an. Der Provider `vminfo` bietet Ihnen die Möglichkeit, mehr über den Ursprung dieser Einlagerungen zu erfahren:

```
dtrace -n pgin'{@[execname] = count()}'
dtrace: description 'pgin' matched 1 probe
^C
  xterm                                                             1
  ksh                                                               1
  ls                                                                2
  lpstat                                                            7
  sh                                                               17
  soffice                                                          39
  javaldx                                                         103
  soffice.bin                                                    3065
```

Die Ausgabe zeigt, dass der mit der StarOffice™-Software verbundene Prozess `soffice.bin` für die meisten Seiteneinlagerungen verantwortlich ist. Um eine genauere Vorstellung von `soffice.bin` in Bezug auf das virtuelle Speicherverhalten zu erhalten, könnten wir nun alle `vminfo`-Prüfpunkte aktivieren. Im nächsten Beispiel wird `dtrace(1M)` beim Starten der StarOffice-Software ausgeführt:

```
dtrace -P vminfo'/execname == "soffice.bin"/{@[probename] = count()}'
dtrace: description 'vminfo' matched 42 probes
^C

  kernel_asflt                                                    1
  fspgin                                                         10
  pgout                                                          16
  execfree                                                       16
  execpgout                                                      16
  fsfree                                                         16
  fspgout                                                        16
  anonfree                                                       16
  anonpgout                                                      16
  pgpgout                                                        16
  dfree                                                          16
  execpgin                                                       80
  prot_fault                                                     85
  maj_fault                                                      88
  pgin                                                           90
  pgpgin                                                         90
  cow_fault                                                     859
  zfod                                                         1619
  pgfrec                                                       8811
  pgrec                                                        8827
  as_fault                                                     9495
```

Das folgende Beispielskript liefert zusätzliche Informationen über das virtuelle Speicherverhalten der StarOffice-Software während des Programmstarts:

```
vminfo:::maj_fault,
vminfo:::zfod,
vminfo:::as_fault
/execname == "soffice.bin" && start == 0/
{
    /*
     * This is the first time that a vminfo probe has been hit; record
     * our initial timestamp.
     */
    start = timestamp;
}

vminfo:::maj_fault,
vminfo:::zfod,
vminfo:::as_fault
/execname == "soffice.bin"/
{
    /*
     * Aggregate on the probename, and lquantize() the number of seconds
```

```
     * since our initial timestamp.  (There are 1,000,000,000 nanoseconds
     * in a second.)  We assume that the script will be terminated before
     * 60 seconds elapses.
     */
    @[probename] =
        lquantize((timestamp - start) / 1000000000, 0, 60);
}
```

Führen Sie das Skript aus, während Sie die StarOffice-Software noch einmal starten. Erstellen Sie dann eine neue Zeichnung und eine neue Präsentation, schließen Sie alle Dateien und beenden Sie die Anwendung. Drücken Sie in der Shell, in der das D-Skript ausgeführt wird, Strg-C. Die Ergebnisse gewähren uns einen Einblick in das virtuelle Speicherverhalten im Verlauf der Zeit.

```
# dtrace -s ./soffice.d
dtrace: script ’./soffice.d’ matched 10 probes
^C

 maj_fault
           value  ------------- Distribution ------------- count
               7 |                                         0
               8 |@@@@@@@@@@                               88
               9 |@@@@@@@@@@@@@@@@@@@@@@@                  194
              10 |@                                        18
              11 |                                         0
              12 |                                         0
              13 |                                         2
              14 |                                         0
              15 |                                         1
              16 |@@@@@@@@@                                82
              17 |                                         0
              18 |                                         0
              19 |                                         2
              20 |                                         0

  zfod
           value  ------------- Distribution ------------- count
             < 0 |                                         0
               0 |@@@@@@@                                  525
               1 |@@@@@@@@                                 605
               2 |@@                                       208
               3 |@@@                                      280
               4 |                                         4
               5 |                                         0
               6 |                                         0
               7 |                                         0
               8 |                                         44
               9 |@@                                       161
```

```
              10 |                                         2
              11 |                                         0
              12 |                                         0
              13 |                                         4
              14 |                                         0
              15 |                                         29
              16 |@@@@@@@@@@@@@@@                          1048
              17 |                                         24
              18 |                                         0
              19 |                                         0
              20 |                                         1
              21 |                                         0
              22 |                                         3
              23 |                                         0

as_fault
         value  ------------- Distribution ------------- count
           < 0 |                                         0
             0 |@@@@@@@@@@@@@@@                          4139
             1 |@@@@@@@                                  2249
             2 |@@@@@@@                                  2402
             3 |@                                        594
             4 |                                         56
             5 |                                         0
             6 |                                         0
             7 |                                         0
             8 |                                         189
             9 |@@                                       929
            10 |                                         39
            11 |                                         0
            12 |                                         0
            13 |                                         6
            14 |                                         0
            15 |                                         297
            16 |@@@@                                     1349
            17 |                                         24
            18 |                                         0
            19 |                                         21
            20 |                                         1
            21 |                                         0
            22 |                                         92
            23 |                                         0
```

Die Ausgabe zeigt einen Teil des StarOffice-Verhaltens in Bezug auf das virtuelle Speichersystem. So wurde beispielsweise der Prüfpunkt `maj_fault` erst ausgelöst, als eine neue Instanz der Anwendung gestartet wurde. Wie zu erhoffen war, hat der „Warmstart“ von StarOffice keine neuen größeren Seitenfehler verursacht. Die Ausgabe von `as_fault` zeigt ein anfängliches Aktivitätshoch, eine gewisse Latenz, während der Benutzer das Menü zum

Erstellen einer neuen Zeichnung sucht, eine weitere Leerlaufzeit und schließlich ein Aktivitätshoch, als der Benutzer auf eine neue Präsentation klickt. Die Ausgabe von `zfod` zeigt, dass das Erstellen der neuen Präsentation über einen kurzen Zeitraum einen starken Bedarf an mit Nullen gefüllten Seiten bewirkt hat.

Der nächste Schritt in der DTrace-Nachforschung für dieses Beispiel hängt von der einzuschlagenden Richtung ab. Wenn Sie verstehen möchten, woher der Bedarf an mit Nullen gefüllten Seiten kommt, könnten Sie `ustack()` in einer `zfod`-Aktivierung aggregieren. Es bietet sich an, einen Schwellwert für die mit Nullen gefüllten Seiten festzulegen und den verletzenden Prozess durch die destruktive Aktion `stop()` anhalten zu lassen, wenn der Schwellwert überschritten wird. Dieser Ansatz würde die Verwendung eher herkömmlicher Debugging-Tools wie `truss(1)` oder `mdb(1)`. Der Provider `vminfo` macht es möglich, die statistischen Daten in den Ausgaben herkömmlicher Tools wie `vmstat(1M)` mit den Anwendungen in Verbindung zu bringen, die das systemische Verhalten verursachen.

Stabilität

Der Provider `vminfo` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Argumente	Private	Private	ISA

◆ ◆ ◆ KAPITEL 25

Der Provider `proc`

Der Provider `proc` stellt Prüfpunkte für die folgenden Aktivitäten zur Verfügung: Erzeugen und Beenden von Prozessen, Erzeugen und Beenden von LWPs, Ausführen neuer Programmabbilder und Senden sowie Behandeln von Signalen.

Prüfpunkte

Die `proc`-Prüfpunkte sind in Tabelle 25–1 beschrieben.

TABELLE 25–1 `proc`-Prüfpunkte

Prüfpunkt	Beschreibung
`create`	Prüfpunkt, der beim Erzeugen eines Prozesses mit `fork(2)`, `forkall(2)`, `fork1(2)` oder `vfork(2)` ausgelöst wird. Auf `psinfo_t` des neuen untergeordneten Prozesses wird mit `args[0]` gezeigt. Sie können `vfork` von den anderen fork-Varianten unterscheiden, indem Sie in der `pr_flag`-Komponente von `lwpsinfo_t` des den neuen Prozess erzeugenden Threads auf `PR_VFORKP` prüfen. Sie können `fork1` von `forkall` unterscheiden, indem Sie die `pr_nlwp`-Komponenten aus `psinfo_t` (`curpsinfo`) des übergeordneten sowie aus `psinfo_t` (`args[0]`) des untergeordneten Prozesses untersuchen. Da der Prüfpunkt `create` erst nach der erfolgreichen Erzeugung des Prozesses ausgelöst wird und die LWP-Erzeugung Teil der Erzeugung eines Prozesses ist, wird `lwp-create` pro LWP ausgelöst, der mit der Prozesserzeugung entsteht, und zwar *bevor* der Prüfpunkt `create` für den neuen Prozess ausgelöst wird.

TABELLE 25–1 proc-Prüfpunkte *(Fortsetzung)*

Prüfpunkt	Beschreibung
exec	Prüfpunkt, der ausgelöst wird, wenn ein Prozess mit einer Variante des exec(2)-Systemaufrufs ein neues Prozessabbild lädt: exec(2), execle(2), execlp(2), execv(2), execve(2), execvp(2). Der Prüfpunkt exec wird ausgelöst, *bevor* das Prozessabbild geladen wird. Prozessvariablen wie execname und curpsinfo enthalten folglich den Prozessstatus vor dem Laden des Abbilds. Kurz nach der Auslösung des Prüfpunkts exec wird in demselben Thread entweder exec-failure oder exec-success ausgelöst. Auf den Pfad des neuen Prozessabbilds zeigt args[0].
exec-failure	Prüfpunkt, der ausgelöst wird, wenn eine exec(2)-Variante fehlschlägt. Der Prüfpunkt exec-failure wird erst nach der Auslösung des Prüfpunkts exec in demselben Thread ausgelöst. Der Wert von errno(3C) steht in args[0] bereit.
exec-success	Prüfpunkt, der ausgelöst wird, wenn eine exec(2)-Variante erfolgreich war. Wie exec-failure wird auch exec-success erst nach der Auslösung des Prüfpunkts exec in demselben Thread ausgelöst. Wenn der Prüfpunkt exec-success ausgelöst wird, enthalten Prozessvariablen wie execname und curpsinfo bereits den Prozessstatus nach dem Laden des neuen Prozessabbilds.
exit	Prüfpunkt, der beim Beenden des aktuellen Prozesses ausgelöst wird. Der Grund für die Beendigung, der in Form einer der SIGCHLD siginfo.h(3HEAD)-Codes ausgedrückt wird, ist in args[0] enthalten.
fault	Prüfpunkt, der ausgelöst wird, wenn in einem Thread ein Maschinenfehler auftritt. Der Fehlercode (gemäß der Definition in proc(4)) befindet sich in args[0]. Auf die Struktur siginfo des Fehlers wird mit args[1] gezeigt. Nur Fehler, die ein Signal auslösen, können auch den Prüfpunkt fault auslösen.
lwp-create	Prüfpunkt, der bei der Erzeugung eines LWP, in der Regel als Folge von thr_create(3C) ausgelöst wird. Auf lwpsinfo_t des neuen Threads wird mit args[0] gezeigt. Auf psinfo_t des Prozesses, der den Thread enthält, wird mit args[1] gezeigt.
lwp-start	Prüfpunkt, der innerhalb des Kontexts eines neu erzeugten LWP ausgelöst wird. Der Prüfpunkt lwp-start wird vor der Ausführung etwaiger Anweisungen auf Benutzerebene ausgelöst. Ist der LWP der erste LWP im Prozess, wird zunächst der Prüfpunkt start und anschließend lwp-start ausgelöst.
lwp-exit	Prüfpunkt, der ausgelöst wird, wenn ein LWP entweder als Reaktion auf ein Signal oder auf einen ausdrücklichen thr_exit(3C)-Aufruf beendet wird.

TABELLE 25–1 proc-Prüfpunkte *(Fortsetzung)*

Prüfpunkt	Beschreibung
signal-discard	Prüfpunkt, der ausgelöst wird, wenn ein Signal an einen Single-Threaded-Prozess gesendet und vom Prozess freigegeben (unblock) und ignoriert wird. Unter diesen Bedingungen wird das Signal bei der Generierung verworfen. lwpsinfo_t und psinfo_t des Zielprozesses und Threads sind in args[0] bzw. args[1] enthalten. Die Signalnummer befindet sich in args[2].
signal-send	Prüfpunkt, der beim Senden eines Signals an einen Thread oder Prozess ausgelöst wird. Der Prüfpunkt signal-send wird im Kontext des sendenden Prozesses und Threads ausgelöst. lwpsinfo_t und psinfo_t des Empfängerprozesses und Threads sind in args[0] bzw. args[1] enthalten. Die Signalnummer befindet sich in args[2]. Auf signal-send folgen im empfangenden Prozess und Thread immer signal-handle oder signal-clear.
signal-handle	Prüfpunkt, der unmittelbar vor der Behandlung eines Signals durch einen Thread ausgelöst wird. Der Prüfpunkt signal-handle wird im Kontext des Threads ausgelöst, der das Signal behandelt. Die Signalnummer befindet sich in args[0]. In args[1] befindet sich ein Zeiger auf die zum Signal gehörige Struktur siginfo_t. Der Wert von args[1] ist NULL, wenn keine siginfo_t-Struktur existiert oder in der Signalbehandlungsroutine das Flag SA_SIGINFO nicht gesetzt ist. Die Adresse des Signal-Handlers im Prozess ist in args[2] enthalten.
signal-clear	Prüfpunkt, der ausgelöst wird, wenn ein anstehendes Signal gelöscht wird, da der Ziel-Thread in sigwait(2), sigwaitinfo(3RT) oder sigtimedwait(3RT) auf das Signal wartete. Unter diesen Bedingungen wird das anstehende Signal gelöscht und die Signalnummer an den Aufrufer zurückgereicht. Die Signalnummer befindet sich in args[0]. signal-clear wird im Kontext des zuvor wartenden Threads ausgelöst.
start	Prüfpunkt, der innerhalb des Kontexts eines neu erzeugten Prozesses ausgelöst wird. Der Prüfpunkt start wird vor der Ausführung etwaiger Anweisungen auf Benutzerebene im Prozess ausgelöst.

Argumente

Die Argumenttypen für proc-Prüfpunkte sind in Tabelle 25–2 aufgeführt. In Tabelle 25–1 finden Sie eine Beschreibung der Argumente.

TABELLE 25–2 Argumente für proc-Prüfpunkte

Prüfpunkt	args[0]	args[1]	args[2]
create	psinfo_t *	—	—

TABELLE 25–2 Argumente für proc-Prüfpunkte *(Fortsetzung)*

Prüfpunkt	args[0]	args[1]	args[2]
exec	char *	—	—
exec-failure	int	—	—
exit	int	—	—
fault	int	siginfo_t *	—
lwp-create	lwpsinfo_t *	psinfo_t *	—
lwp-start	—	—	—
lwp-exit	—	—	—
signal-discard	lwpsinfo_t *	psinfo_t *	int
signal-discard	lwpsinfo_t *	psinfo_t *	int
signal-send	lwpsinfo_t *	psinfo_t *	int
signal-handle	int	siginfo_t *	void (*)(void)
signal-clear	int	—	—
start	—	—	—

lwpsinfo_t

Einige proc-Prüfpunkte besitzen Argumente des Typs lwpsinfo_t, wobei es sich um eine in proc(4) dokumentierte Struktur handelt. Die Definition der den DTrace-Verbrauchern zur Verfügung stehenden Struktur lwpsinfo_t lautet:

```
typedef struct lwpsinfo {
    int pr_flag;              /* flags; see below */
    id_t pr_lwpid;            /* LWP id */
    uintptr_t pr_addr;        /* internal address of thread */
    uintptr_t pr_wchan;       /* wait addr for sleeping thread */
    char pr_stype;            /* synchronization event type */
    char pr_state;            /* numeric thread state */
    char pr_sname;            /* printable character for pr_state */
    char pr_nice;             /* nice for cpu usage */
    short pr_syscall;         /* system call number (if in syscall) */
    int pr_pri;               /* priority, high value = high priority */
    char pr_clname[PRCLSZ];   /* scheduling class name */
    processorid_t pr_onpro;   /* processor which last ran this thread */
    processorid_t pr_bindpro; /* processor to which thread is bound */
    psetid_t pr_bindpset;     /* processor set to which thread is bound */
} lwpsinfo_t;
```

Das Feld `pr_flag` ist eine Bit-Maske, die den Prozess beschreibende Flags enthält. Diese Flags und ihre Bedeutung sind in Tabelle 25–3 beschrieben.

TABELLE 25–3 `pr_flag`-Werte

`PR_ISSYS`	Der Prozess ist ein Systemprozess.
`PR_VFORKP`	Der Prozess ist der übergeordnete Prozess eines `vfork(2)`-Unterprozesses.
`PR_FORK`	Der Modus inherit-on-fork des Prozesses ist gesetzt.
`PR_RLC`	Der Modus run-on-last-close des Prozesses ist gesetzt.
`PR_KLC`	Der Modus kill-on-last-close des Prozesses ist gesetzt.
`PR_ASYNC`	Der Modus asynchronous-stop des Prozesses ist gesetzt.
`PR_MSACCT`	Die Mikrostatusabrechnung (microstate accounting) ist im Prozess aktiviert.
`PR_MSFORK`	Die Mikrostatusabrechnung des Prozesses wird bei fork vererbt.
`PR_BPTADJ`	Der Modus breakpoint-adjustment des Prozesses ist gesetzt.
`PR_PTRACE`	Der Modus `ptrace(3C)`-compatibility des Prozesses ist gesetzt.
`PR_STOPPED`	Der Thread ist ein angehaltener LWP.
`PR_ISTOP`	Der Thread ist ein an einem Ereignis von Interesse angehaltener LWP.
`PR_DSTOP`	Der Thread ist ein LWP, für den eine stop-Direktive in Kraft ist.
`PR_STEP`	Der Thread ist ein LWP, für den eine single-step-Direktive in Kraft ist.
`PR_ASLEEP`	Der Thread ist ein LWP in unterbrechbarem Schlafzustand innerhalb eines Systemaufrufs.
`PR_DETACH`	Der Thread ist ein gesonderter LWP. Informationen hierzu finden Sie unter `pthread_create(3C)` und `pthread_join(3C)`.
`PR_DAEMON`	Der Thread ist ein Dämon-LWP. Siehe hierzu `pthread_create(3C)`.
`PR_AGENT`	Der Thread ist der Agent-LWP für den Prozess.
`PR_IDLE`	Der Thread ist der Leerlauf-Thread für eine CPU. Es laufen nur dann Leerlauf-Threads auf einer CPU, wenn die Ausführungswarteschlangen für die CPU leer sind.

Das Feld `pr_addr` ist die Adresse einer privaten Datenstruktur im Kernel, die den Thread darstellt. Die Datenstruktur ist privat, aber das Feld `pr_addr` kann für die gesamte Lebensdauer eines Threads als eindeutiges Symbol für diesen verwendet werden.

Das Feld `pr_wchan` wird gesetzt, wenn sich der Thread an einem Synchronisierungsobjekt schlafen legt. Die Bedeutung des Felds `pr_wchan` ist der Kernel-Implementierung vorbehalten, aber das Feld kann als eindeutiges Symbol für das Synchronisierungsobjekt verwendet werden.

Das Feld `pr_stype` wird gesetzt, wenn sich der Thread an einem Synchronisierungsobjekt schlafen legt. Die möglichen Werte für das Feld `pr_stype` sehen Sie in Tabelle 25–4.

TABELLE 25–4 `pr_stype`-Werte

`SOBJ_MUTEX`	Mutex-Synchronisierungsobjekt für Kernel. Dient zur Serialisierung des Zugriffs auf gemeinsame Datenbereiche im Kernel. In Kapitel 18, „Der Provider `lockstat`“ und `mutex_init`(9F) finden Sie ausführliche Informationen zu Mutex-Synchronisierungsobjekten für den Kernel.
`SOBJ_RWLOCK`	Leser/Schreiber-Synchronisierungsobjekt für Kernel. Dient zum Synchronisieren des Zugriffs auf gemeinsame Objekte im Kernel, die mehrere Leser gleichzeitig oder einen einzigen Schreiber zulassen können. In ·Kapitel 18, „Der Provider `lockstat`“ und `rwlock`(9F) finden Sie ausführliche Informationen zu Leser/Schreiber-Synchronisierungsobjekten für den Kernel.
`SOBJ_CV`	Bedingungsvariablen-Synchronisierungsobjekt. Eine Bedingungsvariable ist darauf ausgerichtet, auf unbestimmte Zeit darauf zu warten, dass eine bestimmte Bedingung eintritt. Bedingungsvariablen dienen in der Regel zum Synchronisieren anderer Vorgänge als den Zugriffen auf gemeinsame Datenbereiche und stellen den Mechanismus dar, der allgemein verwendet wird, wenn ein Prozess einen programmgesteuerten Wartezustand auf unbestimmte Zeit annimmt. Dabei handelt es sich beispielsweise um das Blockieren in `poll`(2), `pause`(2), `wait`(3C) usw.
`SOBJ_SEMA`	Semaphoren-Synchronisierungsobjekt. Ein Allzweck-Synchronisierungsobjekt, das, wie auch Bedingungsvariablenobjekte, keinen Besitz kennt. Da der Aspekt des Besitzes zum Implementieren der Prioritätsvererbung im Solaris-Kernel erforderlich ist, verbietet das Fehlen dieses Aspekts in Semaphorenobjekten eine weite Verbreitung dieser Objekte. Ausführliche Informationen finden Sie im Abschnitt `semaphore`(9F).
`SOBJ_USER`	Ein Synchronisierungsobjekt auf Benutzerebene Sämtliche Blockierungen von Synchronisierungsobjekten auf Benutzerebene werden mit `SOBJ_USER`-Synchronisierungsobjekten behandelt. Zu den Synchronisierungsobjekten auf Benutzerebene gehören die mit `mutex_init`(3C), `sema_init`(3C), `rwlock_init`(3C), `cond_init`(3C) und deren POSIX-Pendants erzeugten Objekte.
`SOBJ_USER_PI`	Ein Synchronisierungsobjekt auf Benutzerebene, das die Prioritätsvererbung implementiert. Einige Synchronisierungsobjekte auf Benutzerebene, die zusätzliche Benutzerinformationen bieten, ermöglichen die Prioritätsvererbung. So kann beispielsweise für Mutex-Objekte, die mit `pthread_mutex_init`(3C) erzeugt wurden, anhand von `pthread_mutexattr_setprotocol`(3C) die Prioritätsvererbung erreicht werden.
`SOBJ_SHUTTLE`	Ein Shuttle-Synchronisierungsobjekt Shuttle-Objekte dienen zum Implementieren von Doors. Näheres hierzu siehe door_create(3DOOR).

Das Feld `pr_state` wird auf einen der Werte in Tabelle 25–5 gesetzt. In Klammern sehen Sie das entsprechende Zeichen, auf das das Feld `pr_sname` gesetzt wird.

TABELLE 25–5 `pr_state`-Werte

`SSLEEP` (`S`)	Der Thread schläft. Der Prüfpunkt `sched:::sleep` wird ausgelöst, unmittelbar bevor der Status eines Threads in `SSLEEP` übergeht.
`SRUN` (`R`)	Der Thread ist lauffähig, läuft aber derzeit nicht. Der Prüfpunkt `sched:::enqueue` wird ausgelöst, unmittelbar bevor der Status eines Threads in `SRUN` übergeht.
`SZOMB` (`Z`)	Der Thread ist ein Zombie-LWP.
`SSTOP` (`T`)	Der Thread wurde entweder durch eine explizite `proc(4)`-Direktive oder einen anderen Anhaltemechanismus angehalten.
`SIDL` (`I`)	Der Thread befindet sich in einem Zwischenstatus der Prozesserzeugung.
`SONPROC` (`O`)	Der Thread läuft auf einer CPU. Der Prüfpunkt `sched:::on-cpu` wird im Kontext des `SONPROC`-Threads kurz nach dem Übergang des Thread-Status in `SONPROC` ausgelöst.

psinfo_t

Einige `proc`-Prüfpunkte besitzen ein Argument des Typs `psinfo_t`, wobei es sich um eine in `proc(4)` dokumentierte Struktur handelt. Die den DTrace-Verbrauchern zur Verfügung stehende Definition der Struktur `psinfo_t` lautet:

```
typedef struct psinfo {
    int     pr_nlwp;            /* number of active lwps in the process */
    pid_t   pr_pid;             /* unique process id */
    pid_t   pr_ppid;            /* process id of parent */
    pid_t   pr_pgid;            /* pid of process group leader */
    pid_t   pr_sid;             /* session id */
    uid_t   pr_uid;             /* real user id */
    uid_t   pr_euid;            /* effective user id */
    gid_t   pr_gid;             /* real group id */
    gid_t   pr_egid;            /* effective group id */
    uintptr_t pr_addr;          /* address of process */
    dev_t   pr_ttydev;          /* controlling tty device (or PRNODEV) */
    timestruc_t pr_start;       /* process start time, from the epoch */
    char    pr_fname[PRFNSZ];   /* name of execed file */
    char    pr_psargs[PRARGSZ]; /* initial characters of arg list */
    int     pr_argc;            /* initial argument count */
    uintptr_t pr_argv;          /* address of initial argument vector */
    uintptr_t pr_envp;          /* address of initial environment vector */
    char    pr_dmodel;          /* data model of the process */
```

```
    taskid_t pr_taskid;         /* task id */
    projid_t pr_projid;         /* project id */
    poolid_t pr_poolid;         /* pool id */
    zoneid_t pr_zoneid;         /* zone id */
} psinfo_t;
```

Das Feld `pr_dmodel` wird entweder auf `PR_MODEL_ILP32` (bezeichnet einen 32-Bit-Prozess) oder auf `PR_MODEL_LP64` (bezeichnet einen 64-Bit-Prozess) gesetzt.

Beispiele

exec

Der Prüfpunkt exec stellt ein einfaches Mittel dar, um festzustellen, welche Programme von wem ausgeführt werden. Das nächste Beispiel veranschaulicht dies:

```
#pragma D option quiet

proc:::exec
{
        self->parent = execname;
}

proc:::exec-success
/self->parent != NULL/
{
    @[self->parent, execname] = count();
    self->parent = NULL;
}

proc:::exec-failure
/self->parent != NULL/
{
    self->parent = NULL;
}

END
{
    printf("%-20s %-20s %s\n", "WHO", "WHAT", "COUNT");
    printa("%-20s %-20s %@d\n", @);
}
```

Wenn das Beispielskript kurz auf einer Build-Maschine ausgeführt wird, erhalten wir eine ähnliche Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./whoexec.d
^C
WHO                 WHAT                COUNT
make.bin            yacc                1
tcsh                make                1
make.bin            spec2map            1
sh                  grep                1
lint                lint2               1
sh                  lint                1
sh                  ln                  1
cc                  ld                  1
make.bin            cc                  1
lint                lint1               1
sh                  lex                 1
make.bin            mv                  2
sh                  sh                  3
sh                  make                3
sh                  sed                 4
sh                  tr                  4
make                make.bin            4
sh                  install.bin         5
sh                  rm                  6
cc                  ir2hf               33
cc                  ube                 33
sh                  date                34
sh                  mcs                 34
cc                  acomp               34
sh                  cc                  34
sh                  basename            34
basename            expr                34
make.bin            sh                  87
```

`start` und `exit`

Wenn Sie wissen möchten, wie lange Programme zwischen Erzeugung und Ende laufen, können Sie wie folgt die Prüfpunkte `start` und `exit` aktivieren:

```
proc:::start
{
    self->start = timestamp;
}

proc:::exit
/self->start/
{
    @[execname] = quantize(timestamp - self->start);
```

```
    self->start = 0;
}
```

Wenn Sie das Beispielskript einige Sekunden lang auf dem Build-Server ausführen, erhalten Sie eine ähnliche Ausgabe wie diese:

```
# dtrace -s ./progtime.d
dtrace: script ’./progtime.d’ matched 2 probes
^C

  ir2hf
           value  ------------- Distribution ------------- count
         4194304 |                                         0
         8388608 |@                                        1
        16777216 |@@@@@@@@@@@@@@@@@                        14
        33554432 |@@@@@@@@@@                               9
        67108864 |@@@                                      3
       134217728 |@                                        1
       268435456 |@@@@                                     4
       536870912 |@                                        1
      1073741824 |                                         0

  ube
           value  ------------- Distribution ------------- count
        16777216 |                                         0
        33554432 |@@@@@@@                                  6
        67108864 |@@@                                      3
       134217728 |@@                                       2
       268435456 |@@@@                                     4
       536870912 |@@@@@@@@@@@@                             10
      1073741824 |@@@@@@@                                  6
      2147483648 |@@                                       2
      4294967296 |                                         0

  acomp
           value  ------------- Distribution ------------- count
         8388608 |                                         0
        16777216 |@@                                       2
        33554432 |                                         0
        67108864 |@                                        1
       134217728 |@@@                                      3
       268435456 |                                         0
       536870912 |@@@@@                                    5
      1073741824 |@@@@@@@@@@@@@@@@@@@@@@@@@@               22
      2147483648 |@                                        1
      4294967296 |                                         0

  cc
```

```
           value  ------------- Distribution ------------- count
        33554432 |                                         0
        67108864 |@@@                                      3
       134217728 |@                                        1
       268435456 |                                         0
       536870912 |@@@@                                     4
      1073741824 |@@@@@@@@@@@@@@                           13
      2147483648 |@@@@@@@@@@@@                             11
      4294967296 |@@@                                      3
      8589934592 |                                         0

sh
           value  ------------- Distribution ------------- count
          262144 |                                         0
          524288 |@                                        5
         1048576 |@@@@@@@                                  29
         2097152 |                                         0
         4194304 |                                         0
         8388608 |@@@                                      12
        16777216 |@@                                       9
        33554432 |@@                                       9
        67108864 |@@                                       8
       134217728 |@                                        7
       268435456 |@@@@@                                    20
       536870912 |@@@@@@                                   26
      1073741824 |@@@                                      14
      2147483648 |@@                                       11
      4294967296 |                                         3
      8589934592 |                                         1
     17179869184 |                                         0

make.bin
           value  ------------- Distribution ------------- count
        16777216 |                                         0
        33554432 |@                                        1
        67108864 |@                                        1
       134217728 |@@                                       2
       268435456 |                                         0
       536870912 |@@                                       2
      1073741824 |@@@@@@@@@                                9
      2147483648 |@@@@@@@@@@@@@@@                          14
      4294967296 |@@@@@@                                   6
      8589934592 |@@                                       2
     17179869184 |                                         0
```

`lwp-start` und `lwp-exit`

Vielleicht interessiert Sie nicht die Ausführungszeit eines bestimmten Prozesses, sondern einzelner Threads. Das folgende Beispiel zeigt, wie Sie diese Dauer mit den Prüfpunkten `lwp-start` und `lwp-exit` ermitteln können:

```
proc:::lwp-start
/tid != 1/
{
    self->start = timestamp;
}

proc:::lwp-exit
/self->start/
{
    @[execname] = quantize(timestamp - self->start);
    self->start = 0;
}
```

Wenn Sie das Beispielskript auf einem NFS- und Kalenderserver ausführen, erhalten Sie eine ähnliche Ausgabe wie diese:

```
# dtrace -s ./lwptime.d
dtrace: script ’./lwptime.d’ matched 3 probes
^C

  nscd
           value  ------------- Distribution ------------- count
          131072 |                                         0
          262144 |@                                        18
          524288 |@@                                       24
         1048576 |@@@@@@@                                  75
         2097152 |@@@@@@@@@@@@@@@@@@@@@@@@@                245
         4194304 |@@                                       22
         8388608 |@@                                       24
        16777216 |                                         6
        33554432 |                                         3
        67108864 |                                         1
       134217728 |                                         1
       268435456 |                                         0

  mountd
           value  ------------- Distribution ------------- count
          524288 |                                         0
         1048576 |@                                        15
         2097152 |@                                        24
         4194304 |@@@                                      51
         8388608 |@                                        17
```

```
      16777216 |@                                        24
      33554432 |@                                        15
      67108864 |@@@@                                     57
     134217728 |@                                        28
     268435456 |@                                        26
     536870912 |@@                                       39
    1073741824 |@@@                                      45
    2147483648 |@@@@@                                    72
    4294967296 |@@@@@                                    77
    8589934592 |@@@                                      55
   17179869184 |                                         14
   34359738368 |                                         2
   68719476736 |                                         0

automountd
           value  ------------- Distribution ------------- count
       1048576 |                                         0
       2097152 |                                         3
       4194304 |@@@@                                     146
       8388608 |                                         6
      16777216 |                                         6
      33554432 |                                         9
      67108864 |@@@@@                                    203
     134217728 |@@                                       87
     268435456 |@@@@@@@@@@@@@@@                          534
     536870912 |@@@@@@                                   223
    1073741824 |@                                        45
    2147483648 |                                         20
    4294967296 |                                         26
    8589934592 |                                         20
   17179869184 |                                         19
   34359738368 |                                         7
   68719476736 |                                         2
  137438953472 |                                         0

iCald
           value  ------------- Distribution ------------- count
       8388608 |                                         0
      16777216 |@@@@@@@                                  20
      33554432 |@@@                                      9
      67108864 |@@                                       8
     134217728 |@@@@@                                    16
     268435456 |@@@@                                     11
     536870912 |@@@@                                     11
    1073741824 |@                                        4
    2147483648 |                                         2
    4294967296 |                                         0
    8589934592 |@@                                       8
```

```
     17179869184 |@                                        5
     34359738368 |@                                        4
     68719476736 |@@                                       6
    137438953472 |@                                        4
    274877906944 |                                         2
    549755813888 |                                         0
```

signal-send

Mit dem Prüfpunkt `signal-send` lassen sich, wie in folgendem Beispiel demonstriert, der sendende und der empfangende Prozess eines Signals ermitteln:

```
#pragma D option quiet

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

END
{
    printf("%20s %20s %12s %s\n",
        "SENDER", "RECIPIENT", "SIG", "COUNT");
    printa("%20s %20s %12d %@d\n", @);
}
```

Die Ausführung dieses Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./sig.d
^C
              SENDER            RECIPIENT          SIG COUNT
               xterm                dtrace            2 1
               xterm           soffice.bin            2 1
                  tr                 init           18 1
               sched                 test           18 1
               sched                fvwm2           18 1
                bash                 bash           20 1
                  sed                init           18 2
               sched                  ksh           18 15
               sched                 Xsun           22 471
```

Stabilität

Der Provider `proc` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Argumente	Evolving	Evolving	ISA

◆ ◆ ◆ KAPITEL 26

Der Provider sched

Der Provider `sched` stellt Prüfpunkte für die Ablaufplanung in der CPU - das CPU-Scheduling - bereit. Da CPUs die eine Ressource darstellen, die alle Threads verbrauchen müssen, erweist sich der Provider `sched` bei der Untersuchung systemischen Verhaltens als sehr hilfreich. So lässt sich beispielsweise mithilfe des Providers `sched` feststellen, wann und weshalb sich Threads schlafen legen, laufen, ihre Priorität ändern oder andere Threads aufwecken.

Prüfpunkte

Die `sched`-Prüfpunkte sind in Tabelle 26–1 beschrieben.

TABELLE 26–1 `sched`-Prüfpunkte

Prüfpunkt	Beschreibung
`change-pri`	Prüfpunkt, der kurz vor der Änderung der Priorität eines Threads ausgelöst wird. Auf `lwpsinfo_t` des Threads wird mit `args[0]` gezeigt. Die aktuelle Priorität des Threads befindet sich im Feld `pr_pri` dieser Struktur. Auf `psinfo_t` des Prozesses, der den Thread enthält, wird mit `args[1]` gezeigt. Die neue Priorität des Threads ist in `args[2]` enthalten.
`dequeue`	Prüfpunkt, der unmittelbar vor dem Löschen eines ausführbaren Threads aus einer Ausführungswarteschlange ausgelöst wird. Auf `lwpsinfo_t` des Threads, der aus der Warteschlange gelöscht wird, zeigt `args[0]`. Auf `psinfo_t` des Prozesses, der den Thread enthält, wird mit `args[1]` gezeigt. Auf `cpuinfo_t` der CPU, für welche der Thread aus der Warteschlange gelöscht wird, zeigt `args[2]`. Wenn der Thread aus einer Ausführungswarteschlange gelöscht wird, die keiner bestimmten CPU zugeordnet ist, hat die Komponente `cpu_id` dieser Struktur den Wert `-1`.

TABELLE 26–1 sched-Prüfpunkte *(Fortsetzung)*

Prüfpunkt	Beschreibung
enqueue	Prüfpunkt, der ausgeführt wird, unmittelbar bevor ein ausführbarer Thread in eine Ausführungswarteschlange gestellt wird. Auf `lwpsinfo_t` des Threads, der in die Warteschlange gestellt wird, zeigt `args[0]`. Auf `psinfo_t` des Prozesses, der den Thread enthält, wird mit `args[1]` gezeigt. Auf `cpuinfo_t` der CPU, für welche der Thread in die Warteschlange gestellt wird, zeigt `args[2]`. Wenn der Thread in eine Ausführungswarteschlange gestellt wird, die keiner bestimmten CPU zugeordnet ist, hat die Komponente `cpu_id` dieser Struktur den Wert `-1`. `args[3]` enthält einen booleschen Wert, der angibt, ob der Thread an den Anfang der Ausführungswarteschlange gestellt wird. Ist der Wert nicht Null, wird der Thread an den Anfang der Ausführungswarteschlange gestellt, bei Null an das Ende.
off-cpu	Prüfpunkt, der ausgelöst wird, wenn die aktuelle CPU kurz davor steht, die Ausführung eines Threads zu beenden. Die Variable `curcpu` gibt die aktuelle CPU an. Die Variable `curlwpsinfo` steht für den Thread, dessen Ausführung beendet wird. Die Variable `curpsinfo` beschreibt den Prozess, der den aktuellen Thread enthält. Auf die Struktur `lwpsinfo_t` des Threads, den die aktuelle CPU als Nächstes ausführen wird, zeigt `args[0]` . Auf `psinfo_t` des Prozesses, der den nächsten Thread enthält, wird mit `args[1]` gezeigt.
on-cpu	Prüfpunkt, der ausgelöst wird, wenn eine CPU gerade mit der Ausführung eines Threads begonnen hat. Die Variable `curcpu` gibt die aktuelle CPU an. Die Variable `curlwpsinfo` steht für den Thread, dessen Ausführung beginnt. Die Variable `curpsinfo` beschreibt den Prozess, der den aktuellen Thread enthält.
preempt	Prüfpunkt, der ausgelöst wird, unmittelbar bevor der aktuelle Thread unterbrochen wird, weil einem anderen CPU-Zeit zugeteilt wird. Nachdem dieser Prüfpunkt ausgelöst wird, wählt der aktuelle Thread einen auszuführenden Thread, und der Prüfpunkt `off-cpu` wird für den aktuellen Thread ausgelöst. In einigen Fällen wird ein Thread auf einer CPU präemptiv unterbrochen, aber der ihm zuvorkommende Thread wird in der Zwischenzeit auf einer anderen CPU ausgeführt. In diesem Fall wird der Prüfpunkt `preempt` ausgelöst, doch der Dispatcher kann keinen Thread mit höherer Priorität finden, der ausgeführt werden könnte. Dann wird der Prüfpunkt `remain-cpu` anstelle von `off-cpu` ausgelöst.
remain-cpu	Prüfpunkt, der ausgelöst wird, wenn eine Scheduling-Entscheidung getroffen wurde, aber der Dispatcher entschieden hat, den aktuellen Thread weiter auszuführen. Die Variable `curcpu` gibt die aktuelle CPU an. Die Variable `curlwpsinfo` steht für den Thread, dessen Ausführung beginnt. Die Variable `curpsinfo` beschreibt den Prozess, der den aktuellen Thread enthält.

TABELLE 26–1 sched-Prüfpunkte *(Fortsetzung)*

Prüfpunkt	Beschreibung
`schedctl-nopreempt`	Prüfpunkt, der ausgelöst wird, wenn ein Thread präemptiv unterbrochen und aufgrund einer Anforderung der Vorrangunterbrechungssteuerung anschließend wieder an den *Anfang* der Ausführungswarteschlange gestellt wird. `schedctl_init(3C)` enthält weitere Informationen zur Vorrangunterbrechungssteuerung. Ebenso wie bei `preempt` wird nach `schedctl-nopreempt` entweder `off-cpu` oder `remain-cpu` ausgelöst. Da `schedctl-nopreempt` eine Wiedereinreihung des aktuellen Threads an den Anfang der Ausführungswarteschlange kennzeichnet, wird `remain-cpu` wahrscheinlich eher nach `schedctl-nopreempt` ausgelöst als `off-cpu`. Auf `lwpsinfo_t` des Threads, der präemptiv unterbrochen wird, zeigt `args[0]`. Auf `psinfo_t` des Prozesses, der den Thread enthält, wird mit `args[1]` gezeigt.
`schedctl-preempt`	Prüfpunkt, der ausgelöst wird, wenn ein Thread, der mit Vorrangunterbrechungssteuerung arbeitet, trotzdem präemptiv unterbrochen und wieder an das *Ende* der Ausführungswarteschlange gestellt wird. `schedctl_init(3C)` enthält weitere Informationen zur Vorrangunterbrechungssteuerung. Ebenso wie bei `preempt` wird nach `schedctl-preempt` entweder `off-cpu` oder `remain-cpu` ausgelöst. Wie `preempt` (und anders als `schedctl-nopreempt`), kennzeichnet `schedctl-preempt` die Wiedereinreihung des aktuellen Threads am Ende der Ausführungswarteschlange. Folglich wird `off-cpu` mit höherer Wahrscheinlichkeit nach `schedctl-preempt` ausgelöst als `remain-cpu`. Auf `lwpsinfo_t` des Threads, der präemptiv unterbrochen wird, zeigt `args[0]`. Auf `psinfo_t` des Prozesses, der den Thread enthält, wird mit `args[1]` gezeigt.
`schedctl-yield`	Prüfpunkt, der ausgelöst wird, wenn ein Thread mit aktivierter Vorrangunterbrechungssteuerung und einer künstlich verlängerten Zeitscheibe Code ausgeführt hat, um die CPU anderen Threads zu übergeben.
`sleep`	Prüfpunkt, der ausgelöst wird, unmittelbar bevor sich der aktuelle Thread an einem Synchronisierungsobjekt schlafen legt. Der Typ des Synchronisierungsobjekts ist in der Komponente `pr_stype` der Struktur `lwpsinfo_t` enthalten, auf die mit `curlwpsinfo` gezeigt wird . Die Adresse des Synchronisierungsobjekts ist in der Komponente `pr_wchan` der Struktur `lwpsinfo_t` enthalten, auf die mit `curlwpsinfo` gezeigt wird. Die Bedeutung dieser Adresse ist ein privates Implementierungsdetail, aber der Adresswert kann als einmaliges Symbol für das Synchronisierungsobjekt behandelt werden.
`surrender`	Prüfpunkt, der ausgelöst wird, wenn eine CPU von einer anderen CPU angewiesen wird, eine Scheduling-Entscheidung zu treffen - dies häufig, weil ein Thread mit einer höheren Priorität ausführbar geworden ist.

TABELLE 26–1 sched-Prüfpunkte *(Fortsetzung)*

Prüfpunkt	Beschreibung
tick	Prüfpunkt, der im Rahmen einer auf Systemuhr-Ticks basierten Abrechnung ausgelöst wird. Bei der CPU-Abrechnung auf Grundlage von Uhr-Ticks wird untersucht, welche Threads und Prozesse ausgeführt werden, wenn ein Interrupt mit festem Intervall ausgelöst wird. Auf die Struktur `lwpsinfo_t` des Threads, dem CPU-Zeit zugewiesen wird, zeigt `args[0]`. Auf `psinfo_t` des Prozesses, der den Thread enthält, wird mit `args[1]` gezeigt.
wakeup	Prüfpunkt, der ausgelöst wird, unmittelbar bevor der aktuelle Thread einen an einem Synchronisierungsobjekt schlafenden Thread aufweckt. Auf `lwpsinfo_t` des schlafenden Threads zeigt `args[0]`. Auf `psinfo_t` des Prozesses, der den schlafenden Thread enthält, wird mit `args[1]` gezeigt. Der Typ des Synchronisierungsobjekts ist in der Komponente `pr_stype` der Struktur `lwpsinfo_t` des schlafenden Threads enthalten. Die Adresse des Synchronisierungsobjekts ist in der Komponente `pr_wchan` der Struktur `lwpsinfo_t` des schlafenden Threads enthalten. Die Bedeutung dieser Adresse ist ein privates Implementierungsdetail, aber der Adresswert kann als einmaliges Symbol für das Synchronisierungsobjekt behandelt werden.

Argumente

Die Argumenttypen für die sched-Prüfpunkte sind in Tabelle 26–2 aufgeführt. Eine Beschreibung der Argumente finden Sie in Tabelle 26–1.

TABELLE 26–2 Argumente für sched-Prüfpunkte

Prüfpunkt	args[0]	args[1]	args[2]	args[3]
change-pri	lwpsinfo_t *	psinfo_t *	pri_t	—
dequeue	lwpsinfo_t *	psinfo_t *	cpuinfo_t *	—
enqueue	lwpsinfo_t *	psinfo_t *	cpuinfo_t *	int
off-cpu	lwpsinfo_t *	psinfo_t *	—	—
on-cpu	—	—	—	—
preempt	—	—	—	—
remain-cpu	—	—	—	—
schedctl-nopreempt	lwpsinfo_t *	psinfo_t *	—	—
schedctl-preempt	lwpsinfo_t *	psinfo_t *	—	—
schedctl-yield	lwpsinfo_t *	psinfo_t *	—	—

TABELLE 26–2 Argumente für sched-Prüfpunkte *(Fortsetzung)*

Prüfpunkt	args[0]	args[1]	args[2]	args[3]
sleep	—	—	—	—
surrender	lwpsinfo_t *	psinfo_t *	—	—
tick	lwpsinfo_t *	psinfo_t *	—	—
wakeup	lwpsinfo_t *	psinfo_t *	—	—

Wie aus Tabelle 26–2 hervorgeht, bestehen die Argumente vieler sched-Prüfpunkte aus einem Zeiger auf eine Struktur lwpsinfo_t und einem Zeiger auf eine Struktur psinfo_t, die für einen Thread und den den Thread enthaltenden Prozess stehen. Diese Strukturen werden ausführlich in „lwpsinfo_t“ auf Seite 266 und „psinfo_t“ auf Seite 269 beschrieben.

cpuinfo_t

Die Struktur cpuinfo_t definiert eine CPU. Wie Sie in Tabelle 26–2 sehen, haben sowohl der Prüfpunkt enqueue als auch dequeue einen Zeiger auf eine Struktur cpuinfo_t als Argument. Zusätzlich wird auf die der aktuellen CPU entsprechenden Struktur cpuinfo_t mit der Variable curcpu gezeigt. Die Definition der Struktur cpuinfo_t lautet:

```
typedef struct cpuinfo {
    processorid_t cpu_id;           /* CPU identifier */
    psetid_t cpu_pset;              /* processor set identifier */
    chipid_t cpu_chip;              /* chip identifier */
    lgrp_id_t cpu_lgrp;             /* locality group identifer */
    processor_info_t cpu_info;      /* CPU information */
} cpuinfo_t;
```

Die Komponente cpu_id ist die Prozessor-ID, wie sie psrinfo(1M) und p_online(2) zurückgeben.

Die Komponente cpu_pset ist ggf. der Prozessorsatz, der die CPU enthält. psrset(1M) enthält weitere Informationen über Prozessorsätze.

Die Komponente cpu_chip ist die ID des physischen Chips. Physische Chips können mehrere CPUs enthalten. Weitere Informationen finden Sie unter psrinfo(1M).

Die Komponente cpu_lgrp ist die ID der Latenzgruppe der CPU. Unter liblgrp(3LIB) finden Sie ausführliche Informationen zu Latenzgruppen.

Die Komponente cpu_info ist die Struktur processor_info_t für die CPU, wie sie von processor_info(2) zurückgegeben wird.

Beispiele

on-cpu und off-cpu

Welche CPUs führen Threads aus und wie lange? Dies ist eine der Fragen, auf die man häufig gerne eine Antwort hätte. Mit den Prüfpunkten on-cpu und off-cpu lässt sich diese Frage, wie das folgende Beispiel zeigt, auf systemweiter Basis einfach beantworten:

```
sched:::on-cpu
{
    self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
    @[cpu] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

Die Ausführung des obigen Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./where.d
dtrace: script ’./where.d’ matched 5 probes
^C

        0
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@@                                       37
            8192 |@@@@@@@@@@@@@@                           212
           16384 |@                                        30
           32768 |                                         10
           65536 |@                                        17
          131072 |                                         12
          262144 |                                         9
          524288 |                                         6
         1048576 |                                         5
         2097152 |                                         1
         4194304 |                                         3
         8388608 |@@@@                                     75
        16777216 |@@@@@@@@@@@@@@                           201
        33554432 |                                         6
        67108864 |                                         0

        1
```

```
           value  ------------- Distribution ------------- count
            2048 |                                         0
            4096 |@                                        6
            8192 |@@@@                                     23
           16384 |@@@                                      18
           32768 |@@@@                                     22
           65536 |@@@@                                     22
          131072 |@                                        7
          262144 |                                         5
          524288 |                                         2
         1048576 |                                         3
         2097152 |@                                        9
         4194304 |                                         4
         8388608 |@@@                                      18
        16777216 |@@@                                      19
        33554432 |@@@                                      16
        67108864 |@@@@                                     21
       134217728 |@@                                       14
       268435456 |                                         0
```

Die obige Ausgabe zeigt, dass auf CPU 1 Threads über einen Zeitraum von weniger als 100 Mikrosekunden oder für ca. 10 Millisekunden laufen. Zwischen den zwei Datengruppen ist im Histogramm eine deutliche Lücke zu erkennen. Sie möchten vielleicht auch wissen, welche CPUs einen bestimmten Prozess ausführen. Auch diese Frage können Sie mit den Prüfpunkten `on-cpu` und `off-cpu` beantworten. Das folgende Skript zeigt, welche CPUs eine angegebene Anwendung über einen Zeitraum von zehn Sekunden ausführen:

```
#pragma D option quiet

dtrace:::BEGIN
{
    start = timestamp;
}

sched:::on-cpu
/execname == $$1/
{
    self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
    @[cpu] = sum(timestamp - self->ts);
    self->ts = 0;
}

profile:::tick-1sec
```

```
/++x == 10/
{
    exit(0);
}

dtrace:::END
{
    printf("CPU distribution of imapd over %d seconds:\n\n",
        (timestamp - start) / 1000000000);
    printf("CPU microseconds\n--- ------------\n");
    normalize(@, 1000);
    printa("%3d %@d\n", @);
}
```

Wenn Sie dieses Skript auf einem großen Mail-Server ausführen und den IMAP-Dämon angeben, erhalten Sie eine ähnliche Ausgabe wie diese:

```
# dtrace -s ./whererun.d imapd
CPU distribution of imapd over 10 seconds:

CPU microseconds
--- ------------
 15 10102
 12 16377
 21 25317
 19 25504
 17 35653
 13 41539
 14 46669
 20 57753
 22 70088
 16 115860
 23 127775
 18 160517
```

In Solaris wird bei der Auswahl einer CPU, auf der Threads ausgeführt werden sollen, die Schlafzeit der Threads berücksichtigt: Threads, die seit kürzerer Zeit schlafen, migrieren in der Regel eher nicht. Mit den Prüfpunkten `on-cpu` und `off-cpu` lässt sich dieses Verhalten beobachten:

```
sched:::off-cpu
/curlwpsinfo->pr_state == SSLEEP/
{
    self->cpu = cpu;
    self->ts = timestamp;
}

sched:::on-cpu
```

```
/self->ts/
{
    @[self->cpu == cpu ?
        "sleep time, no CPU migration" : "sleep time, CPU migration"] =
        lquantize((timestamp - self->ts) / 1000000, 0, 500, 25);
    self->ts = 0;
    self->cpu = 0;
}
```

Wenn Sie das obige Skript ungefähr 30 Sekunden lang ausführen, erhalten Sie eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./howlong.d
dtrace: script './howlong.d' matched 5 probes
^C
 sleep time, CPU migration
           value  -------------- Distribution ------------ count
             < 0 |                                         0
               0 |@@@@@@@                                  6838
              25 |@@@@@                                    4714
              50 |@@@                                      3108
              75 |@                                        1304
             100 |@                                        1557
             125 |@                                        1425
             150 |                                         894
             175 |@                                        1526
             200 |@@                                       2010
             225 |@@                                       1933
             250 |@@                                       1982
             275 |@@                                       2051
             300 |@@                                       2021
             325 |@                                        1708
             350 |@                                        1113
             375 |                                         502
             400 |                                         220
             425 |                                         106
             450 |                                         54
             475 |                                         40
         >= 500 |@                                        1716

  sleep time, no CPU migration
           value  -------------- Distribution ------------ count
             < 0 |                                         0
               0 |@@@@@@@@@@@@@                            58413
              25 |@@@                                      14793
              50 |@@                                       10050
              75 |                                         3858
             100 |@                                        6242
```

```
            125 |@                                        6555
            150 |                                         3980
            175 |@                                        5987
            200 |@                                        9024
            225 |@                                        9070
            250 |@@                                       10745
            275 |@@                                       11898
            300 |@@                                       11704
            325 |@@                                       10846
            350 |@                                        6962
            375 |                                         3292
            400 |                                         1713
            425 |                                         585
            450 |                                         201
            475 |                                         96
          >= 500 |                                         3946
```

Aus der Beispielausgabe geht hervor, dass sehr viel häufiger keine Migration stattfindet. Auch zeigt sie, dass eine Migration bei längeren Schlafzeiten wahrscheinlicher ist. Die Verteilungen im Bereich unter 100 Millisekunden fallen deutlich unterschiedlich aus, gleichen sich aber mit zunehmender Schlafdauer immer näher an. Dieses Ergebnis könnte man dahin gehend interpretieren, dass die Schlafzeit bei der Scheduling-Entscheidung keine Rolle mehr spielt, sobald ein bestimmter Schwellwert überschritten ist.

Unser letztes Beispiel für die Prüfpunkte `off-cpu` und `on-cpu` demonstriert deren Einsatz gemeinsam mit dem Feld `pr_stype`, um festzustellen, weshalb und wie lange Threads schlafen:

```
sched:::off-cpu
/curlwpsinfo->pr_state == SSLEEP/
{
    /*
     * We're sleeping.  Track our sobj type.
     */
    self->sobj = curlwpsinfo->pr_stype;
    self->bedtime = timestamp;
}

sched:::off-cpu
/curlwpsinfo->pr_state == SRUN/
{
    self->bedtime = timestamp;
}

sched:::on-cpu
/self->bedtime && !self->sobj/
{
    @["preempted"] = quantize(timestamp - self->bedtime);
    self->bedtime = 0;
```

```
}

sched:::on-cpu
/self->sobj/
{
    @[self->sobj == SOBJ_MUTEX ? "kernel-level lock" :
        self->sobj == SOBJ_RWLOCK ? "rwlock" :
        self->sobj == SOBJ_CV ? "condition variable" :
        self->sobj == SOBJ_SEMA ? "semaphore" :
        self->sobj == SOBJ_USER ? "user-level lock" :
        self->sobj == SOBJ_USER_PI ? "user-level prio-inheriting lock" :
        self->sobj == SOBJ_SHUTTLE ? "shuttle" : "unknown"] =
        quantize(timestamp - self->bedtime);

    self->sobj = 0;
    self->bedtime = 0;
}
```

Wenn Sie das obige Skript einige Sekunden lang ausführen, erhalten Sie eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./whatfor.d
dtrace: script ’./whatfor.d’ matched 12 probes
^C
 kernel-level lock
           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@@@@@                                 3
           65536 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  11
          131072 |@@                                       1
          262144 |                                         0

  preempted
           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |                                         4
           65536 |@@@@@@@@                                 408
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  1031
          262144 |@@@                                      156
          524288 |@@                                       116
         1048576 |@                                        51
         2097152 |                                         42
         4194304 |                                         16
         8388608 |                                         15
        16777216 |                                         4
        33554432 |                                         8
        67108864 |                                         0
```

```
semaphore
           value  ------------- Distribution ------------ count
           32768 |                                         0
           65536 |@@                                       61
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@               553
          262144 |@@                                       63
          524288 |@                                        36
         1048576 |                                         7
         2097152 |                                         22
         4194304 |@                                        44
         8388608 |@@@                                      84
        16777216 |@                                        36
        33554432 |                                         3
        67108864 |                                         6
       134217728 |                                         0
       268435456 |                                         0
       536870912 |                                         0
      1073741824 |                                         0
      2147483648 |                                         0
      4294967296 |                                         0
      8589934592 |                                         0
     17179869184 |                                         1
     34359738368 |                                         0

shuttle
           value  ------------- Distribution ------------ count
           32768 |                                         0
           65536 |@@@@@                                    2
          131072 |@@@@@@@@@@@@@@@@@                        6
          262144 |@@@@@                                    2
          524288 |                                         0
         1048576 |                                         0
         2097152 |                                         0
         4194304 |@@@@@                                    2
         8388608 |                                         0
        16777216 |                                         0
        33554432 |                                         0
        67108864 |                                         0
       134217728 |                                         0
       268435456 |                                         0
       536870912 |                                         0
      1073741824 |                                         0
      2147483648 |                                         0
      4294967296 |@@@@@                                    2
      8589934592 |                                         0
     17179869184 |@@                                       1
     34359738368 |                                         0
```

```
  condition variable
           value  -------------- Distribution ------------ count
            32768 |                                         0
            65536 |                                         122
           131072 |@@@@@                                    1579
           262144 |@                                        340
           524288 |                                         268
          1048576 |@@@                                      1028
          2097152 |@@@                                      1007
          4194304 |@@@                                      1176
          8388608 |@@@@                                     1257
         16777216 |@@@@@@@@@@@@@@                           4385
         33554432 |                                         295
         67108864 |                                         157
        134217728 |                                         96
        268435456 |                                         48
        536870912 |                                         144
       1073741824 |                                         10
       2147483648 |                                         22
       4294967296 |                                         18
       8589934592 |                                         5
      17179869184 |                                         6
      34359738368 |                                         4
      68719476736 |                                         0
```

enqueue **und** dequeue

Wenn eine CPU in den Ruhezustand übergeht, sucht der Dispatcher nach Arbeit in den Warteschlangen anderer (belegter) CPUs. Im nächsten Beispiel soll anhand des Prüfpunkts dequeue beobachtet werden, wie oft und von welchen CPUs Anwendungen verschoben werden:

```
#pragma D option quiet

sched:::dequeue
/args[2]->cpu_id != --1 && cpu != args[2]->cpu_id &&
    (curlwpsinfo->pr_flag & PR_IDLE)/
{
    @[stringof(args[1]->pr_fname), args[2]->cpu_id] =
        lquantize(cpu, 0, 100);
}

END
{
    printa("%s stolen from CPU %d by:\n%@d\n", @);
}
```

Der letzte Teil der Ausgabe, die Sie erhalten, wenn Sie dieses Skript auf einem System mit vier CPUs ausführen, lautet etwa:

```
# dtrace -s ./whosteal.d
^C
...
 nscd stolen from CPU 1 by:

           value  ------------- Distribution ------------ count
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 28
               3 |                                         0

snmpd stolen from CPU 1 by:

           value  ------------- Distribution ------------ count
             < 0 |                                         0
               0 |@                                        1
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     31
               3 |@@                                       2
               4 |                                         0

sched stolen from CPU 1 by:

           value  ------------- Distribution ------------ count
             < 0 |                                         0
               0 |@@                                       3
               1 |                                         0
               2 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      36
               3 |@@@@                                     5
               4 |                                         0
```

Anstatt festzustellen, welche CPUs welche Arbeit übernommen haben, möchten Sie aber möglicherweise wissen, auf welchen CPUs Prozesse und Threads auf ihre Ausführung warten. Diese Frage können wir mit einer Kombination aus den Prüfpunkten enqueue und dequeue beantworten:

```
sched:::enqueue
{
    self->ts = timestamp;
}

sched:::dequeue
/self->ts/
{
    @[args[2]->cpu_id] = quantize(timestamp - self->ts);
```

```
    self->ts = 0;
}
```

Wenn Sie das obige Skript einige Sekunden lang ausführen, erhalten Sie eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./qtime.d
dtrace: script ’./qtime.d’ matched 5 probes
^C

       -1
           value  -------------- Distribution ------------ count
            4096 |                                         0
            8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
           16384 |                                         0

        0
           value  -------------- Distribution ------------ count
            1024 |                                         0
            2048 |@@@@@@@@@@@@@@@@                         262
            4096 |@@@@@@@@@@@@@@                          227
            8192 |@@@@@                                   87
           16384 |@@@                                     54
           32768 |                                         7
           65536 |                                         9
          131072 |                                         1
          262144 |                                         5
          524288 |                                         4
         1048576 |                                         2
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         0
        16777216 |                                         1
        33554432 |                                         2
        67108864 |                                         2
       134217728 |                                         0
       268435456 |                                         0
       536870912 |                                         0
      1073741824 |                                         1
      2147483648 |                                         1
      4294967296 |                                         0

        1
           value  -------------- Distribution ------------ count
            1024 |                                         0
            2048 |@@@@                                    49
            4096 |@@@@@@@@@@@@@@@@@@@@                    241
            8192 |@@@@@@@                                 91
           16384 |@@@@                                    55
```

```
          32768 |                                         7
          65536 |                                         3
         131072 |                                         2
         262144 |                                         1
         524288 |                                         0
        1048576 |                                         0
        2097152 |                                         0
        4194304 |                                         0
        8388608 |                                         0
       16777216 |                                         0
       33554432 |                                         3
       67108864 |                                         1
      134217728 |                                         4
      268435456 |                                         2
      536870912 |                                         0
     1073741824 |                                         3
     2147483648 |                                         2
     4294967296 |                                         0
```

Beachten Sie die Werte ungleich Null am Ende der Beispielausgabe. Wie diese Datenpunkte zeigen, kam es auf beiden CPUs mehrmals vor, dass ein Thread mehrere *Sekunden* lang in der Ausführungswarteschlange stand.

Möglicherweise interessiert Sie aber nicht so sehr die jeweilige Wartezeit, sondern die Länge der Ausführungswarteschlange im Verlauf der Zeit. Mit den Prüfpunkten `enqueue` und `dequeue` können Sie einen assoziativen Vektor zum Ermitteln der Warteschlangenlänge einrichten:

```
sched:::enqueue
{
    this->len = qlen[args[2]->cpu_id]++;
    @[args[2]->cpu_id] = lquantize(this->len, 0, 100);
}

sched:::dequeue
/qlen[args[2]->cpu_id]/
{
    qlen[args[2]->cpu_id]–;
}
```

Wenn Sie das obige Skript ungefähr 30 Sekunden lang auf einem großteilig im Ruhezustand befindlichen Laptop-System ausführen, erhalten Sie eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./qlen.d
dtrace: script ’./qlen.d’ matched 5 probes
^C
        0
           value  ------------- Distribution ------------ count
             < 0 |                                         0
```

```
     0 |@@@@@@@@@@@@@@@@@@@@@@@@@@                 110626
     1 |@@@@@@@@@@                                 41142
     2 |@@                                         12655
     3 |@                                          5074
     4 |                                           1722
     5 |                                           701
     6 |                                           302
     7 |                                           63
     8 |                                           23
     9 |                                           12
    10 |                                           24
    11 |                                           58
    12 |                                           14
    13 |                                           3
    14 |                                           0
```

Die Ausgabe entspricht grob den allgemeinen Erwartungen für ein System im Ruhezustand: Wenn ein ausführbarer Thread in die Warteschlange gestellt wird, ist die Warteschlange meistens sehr kurz (drei oder weniger Threads lang). Da sich aber das System großteilig im Ruhezustand befand, fallen die außergewöhnlichen Datenpunkte am Ende der Tabelle als eher unerwartet auf. Weshalb war zum Beispiel die Ausführungswarteschlange länger als 13 ausführbare Threads? Um dieser Frage auf den Grund zu gehen, könnten wir ein D-Skript schreiben, das den Inhalt der Ausführungswarteschlange anzeigt, wenn diese eine gewisse Länge erreicht. Das Problem ist allerdings recht komplex, da D-Aktivierungen nicht über Datenstrukturen wiederholt und folglich nicht einfach über der gesamten Ausführungswarteschlange wiederholt werden können. Selbst wenn dies möglich wäre, müssten Abhängigkeiten von den kernelinternen Datenstrukturen vermieden werden.

Für diese Art Skript würden wir die Prüfpunkte `enqueue` und `dequeue` aktivieren und sowohl Spekulationen als auch assoziative Vektoren verwenden. Immer wenn ein Thread in die Warteschlange gestellt wird, erhöht das Skript die Länge der Warteschlange und zeichnet die Zeitmarke in einem assoziativen Vektor mit Schlüssel des Threads auf. In diesem Fall kann keine thread-lokale Variable verwendet werden, da ein Thread auch durch einen anderen Thread in die Warteschlange gestellt werden kann. Das Skript überprüft dann, ob die Warteschlangenlänge das Maximum übersteigt. ist dies der Fall, beginnt das Skript eine neue Spekulation und zeichnet die Zeitmarke und das neue Maximum auf. Wenn dann ein Thread aus der Warteschlange gelöscht wird, vergleicht das Skript die Zeitmarke der Einreihung in die Warteschlange mit derjenigen der maximalen Länge: Wenn der Thread *vor* der Zeitmarke der maximalen Länge in die Warteschlange gestellt wurde, befand er sich bereits in der Warteschlange, als die maximale Länge aufgezeichnet wurde. In diesem Fall führt das Skript eine spekulative Ablaufverfolgung der Thread-Informationen durch. Wenn der Kernel den letzten zum Zeitpunkt der maximalen Länge eingereihten Thread aus der Warteschlange löscht, übergibt das Skript die Spekulationsdaten. Das Skript sieht wie folgt aus:

```
#pragma D option quiet
#pragma D option nspec=4
#pragma D option specsize=100k
```

```
int maxlen;
int spec[int];

sched:::enqueue
{
    this->len = ++qlen[this->cpu = args[2]->cpu_id];
    in[args[0]->pr_addr] = timestamp;
}

sched:::enqueue
/this->len > maxlen && spec[this->cpu]/
{
    /*
     * There is already a speculation for this CPU.  We just set a new
     * record, so we'll discard the old one.
     */
    discard(spec[this->cpu]);
}

sched:::enqueue
/this->len > maxlen/
{
    /*
     * We have a winner.  Set the new maximum length and set the timestamp
     * of the longest length.
     */
    maxlen = this->len;
    longtime[this->cpu] = timestamp;

    /*
     * Now start a new speculation, and speculatively trace the length.
     */
    this->spec = spec[this->cpu] = speculation();
    speculate(this->spec);
    printf("Run queue of length %d:\n", this->len);
}

sched:::dequeue
/(this->in = in[args[0]->pr_addr]) &&
    this->in <= longtime[this->cpu = args[2]->cpu_id]/
{
    speculate(spec[this->cpu]);
    printf("  %d/%d (%s)\n",
        args[1]->pr_pid, args[0]->pr_lwpid,
        stringof(args[1]->pr_fname));
}
```

```
sched:::dequeue
/qlen[args[2]->cpu_id]/
{
    in[args[0]->pr_addr] = 0;
    this->len = --qlen[args[2]->cpu_id];
}

sched:::dequeue
/this->len == 0 && spec[this->cpu]/
{
    /*
     * We just processed the last thread that was enqueued at the time
     * of longest length; commit the speculation, which by now contains
     * each thread that was enqueued when the queue was longest.
     */
    commit(spec[this->cpu]);
    spec[this->cpu] = 0;
}
```

Die Ausführung des obigen Skripts auf demselben Einzelprozessor-Laptop erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./whoqueue.d
Run queue of length 3:
 0/0 (sched)
  0/0 (sched)
  101170/1 (dtrace)
Run queue of length 4:
  0/0 (sched)
  100356/1 (Xsun)
  100420/1 (xterm)
  101170/1 (dtrace)
Run queue of length 5:
  0/0 (sched)
  0/0 (sched)
  100356/1 (Xsun)
  100420/1 (xterm)
  101170/1 (dtrace)
Run queue of length 7:
  0/0 (sched)
  100221/18 (nscd)
  100221/17 (nscd)
  100221/16 (nscd)
  100221/13 (nscd)
  100221/14 (nscd)
  100221/15 (nscd)
Run queue of length 16:
  100821/1 (xterm)
```

```
  100768/1 (xterm)
  100365/1 (fvwm2)
  101118/1 (xterm)
  100577/1 (xterm)
  101170/1 (dtrace)
  101020/1 (xterm)
  101089/1 (xterm)
  100795/1 (xterm)
  100741/1 (xterm)
  100710/1 (xterm)
  101048/1 (xterm)
  100697/1 (MozillaFirebird-)
  100420/1 (xterm)
  100394/1 (xterm)
  100368/1 (xterm)
^C
```

Die Ausgabe zeigt auf, dass die langen Ausführungswarteschlangen auf viele ausführbare `xterm`-Prozesse zurückzuführen sind. Dieses Experiment fiel mit einem Wechsel des virtuellen Desktops zusammen. Die Ergebnisse lassen sich deshalb wahrscheinlich durch eine Art der Verarbeitung von X-Ereignissen erklären.

`sleep` und `wakeup`

Das letzte Beispiel unter „enqueue und dequeue“ auf Seite 291 hat gezeigt, dass ausführbare `xterm`-Prozesse zu besonders langen Ausführungswarteschlangen geführt haben. Es ist denkbar, dass diese Beobachtung auf einen Wechsel des virtuellen Desktops zurückzuführen ist. Mit dem Prüfpunkt `wakeup` können Sie diese Hypothese untersuchen, indem Sie ermitteln, wer die `xterm`-Prozesse wann aufweckt:

```
#pragma D option quiet

dtrace:::BEGIN
{
    start = timestamp;
}

sched:::wakeup
/stringof(args[1]->pr_fname) == "xterm"/
{
    @[execname] = lquantize((timestamp - start) / 1000000000, 0, 10);
}

profile:::tick-1sec
/++x == 10/
{
```

```
    exit(0);
}
```

Um dieser Frage nachzugehen, führen Sie also das obige Skript aus, warten rund fünf Sekunden und wechseln den Desktop genau einmal. Wenn das hohe Aufkommen ausführbarer `xterm`-Prozesse auf den Wechsel des virtuellen Desktops zurückzuführen ist, sollte die Ausgabe an der 5-Sekunden-Marke eine hohe Aufweck-Aktivität zeigen.

```
# dtrace -s ./xterm.d

  Xsun

           value  -------------- Distribution ------------ count
               4 |                                         0
               5 |@                                        1
               6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   32
               7 |                                         0
```

In der Ausgabe ist zu sehen, dass der X-Server zu dem Zeitpunkt, zu dem Sie den virtuellen Desktop gewechselt haben, gehäuft xterm-Prozesse aufweckt. Wenn Sie die Interaktion zwischen dem X-Server und den `xterm`-Prozessen erkunden möchten, könnten Sie die Benutzer-Stack-Aufzeichnungen aggregieren, wenn der X-Server den Prüfpunkt `wakeup` auslöst.

Um Client-Server-Systeme wie das X-Fenstersystem zu verstehen, muss man zunächst die Clients verstehen, für die der Server arbeitet. Diese Frage ist mit herkömmlichen Tools für die Leistungsanalyse nicht leicht zu beantworten. Bei Modellen, in welchen ein Client eine Meldung an den Server sendet und in Erwartung der Verarbeitung durch den Server schläft, können Sie jedoch den Prüfpunkt `wakeup` einsetzen, um den Client zu ermitteln, für den die Anforderung läuft. Das folgende Beispiel zeigt dies:

```
self int last;

sched:::wakeup
/self->last && args[0]->pr_stype == SOBJ_CV/
{
    @[stringof(args[1]->pr_fname)] = sum(vtimestamp - self->last);
    self->last = 0;
}

sched:::wakeup
/execname == "Xsun" && self->last == 0/
{
    self->last = vtimestamp;
}
```

Die Ausführung des obigen Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
dtrace -s ./xwork.d
dtrace: script './xwork.d' matched 14 probes
^C
  xterm                                                     9522510
  soffice.bin                                               9912594
  fvwm2                                                   100423123
  MozillaFirebird                                         312227077
  acroread                                                345901577
```

An dieser Ausgabe erkennen wir, dass ein Großteil der `Xsun`-Arbeit für die Prozesse `acroread`, `MozillaFirebird` und in geringerem Maße auch für `fvwm2` geleistet wird. Beachten Sie, dass das Skript nur wakeups von Synchronisierungsobjekten in Form von Bedingungsvariablen (`SOBJ_CV`) untersucht. Wie aus Tabelle 25–4 hervorgeht, werden Bedingungsvariablen in der Regel dort als Synchronisierungsobjekte eingesetzt, wo aus anderen Gründen als einem Zugriff auf einen gemeinsam genutzten Datenbereich eine Synchronisierung erforderlich ist. Im Fall des X-Servers wartet ein Client auf Daten in einer Pipe, indem er sich an einer Bedingungsvariable schlafen legt.

Zusätzlich können Sie wie im nächsten Beispiel den Prüfpunkt `sleep` neben `wakeup` verwenden, um nachzuvollziehen, welche Anwendungen durch welche anderen Anwendungen wie lang blockiert werden:

```
#pragma D option quiet

sched:::sleep
/!(curlwpsinfo->pr_flag & PR_ISSYS) && curlwpsinfo->pr_stype == SOBJ_CV/
{
    bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
    @[stringof(args[1]->pr_fname), execname] =
        quantize(timestamp - bedtime[args[0]->pr_addr]);
    bedtime[args[0]->pr_addr] = 0;
}

END
{
    printa("%s sleeping on %s:\n%@d\n", @);
}
```

Das Ende der Ausgabe, die Sie erhalten, wenn Sie das Beispielskript einige Sekunden auf einem Desktop-System ausführen, sieht ungefähr wie folgt aus:

```
# dtrace -s ./whofor.d
^C
```

```
...
 xterm sleeping on Xsun:

           value  -------------- Distribution ------------ count
          131072 |                                         0
          262144 |                                         12
          524288 |                                         2
         1048576 |                                         0
         2097152 |                                         5
         4194304 |@@@                                      45
         8388608 |                                         1
        16777216 |                                         9
        33554432 |@@@@@                                    83
        67108864 |@@@@@@@@@@                               164
       134217728 |@@@@@@@@@                                147
       268435456 |@@@@                                     56
       536870912 |@                                        17
      1073741824 |                                         9
      2147483648 |                                         1
      4294967296 |                                         3
      8589934592 |                                         1
     17179869184 |                                         0

fvwm2 sleeping on Xsun:

           value  -------------- Distribution ------------ count
           32768 |                                         0
           65536 |@@@@@@@@@@@@@@@@@@@@@@                   67
          131072 |@@@@@                                    16
          262144 |@@                                       6
          524288 |@                                        3
         1048576 |@@@@@                                    15
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         1
        16777216 |                                         0
        33554432 |                                         0
        67108864 |                                         1
       134217728 |                                         0
       268435456 |                                         0
       536870912 |                                         1
      1073741824 |                                         1
      2147483648 |                                         2
      4294967296 |                                         2
      8589934592 |                                         2
     17179869184 |                                         0
     34359738368 |                                         2
     68719476736 |                                         0
```

```
syslogd sleeping on syslogd:

           value  ------------- Distribution ------------ count
     17179869184 |                                         0
     34359738368 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3
     68719476736 |                                         0

MozillaFirebird sleeping on MozillaFirebird:

           value  ------------- Distribution ------------ count
           65536 |                                         0
          131072 |                                         3
          262144 |@@                                       14
          524288 |                                         0
         1048576 |@@@                                      18
         2097152 |                                         0
         4194304 |                                         0
         8388608 |                                         1
        16777216 |                                         0
        33554432 |                                         1
        67108864 |                                         3
       134217728 |@                                        7
       268435456 |@@@@@@@@@@                               53
       536870912 |@@@@@@@@@@@@@@@                          78
      1073741824 |@@@@                                     25
      2147483648 |                                         0
      4294967296 |                                         0
      8589934592 |@                                        7
     17179869184 |                                         0
```

Es könnte interessant sein, zu verstehen, wie und weshalb `MozillaFirebird` durch sich selbst blockiert wird. Um diese Frage zu beantworten, ändern Sie das obige Skript wie folgt ab:

```
#pragma D option quiet

sched:::sleep
/execname == "MozillaFirebird" && curlwpsinfo->pr_stype == SOBJ_CV/
{
    bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/execname == "MozillaFirebird" && bedtime[args[0]->pr_addr]/
{
    @[args[1]->pr_pid, args[0]->pr_lwpid, pid, curlwpsinfo->pr_lwpid] =
        quantize(timestamp - bedtime[args[0]->pr_addr]);
    bedtime[args[0]->pr_addr] = 0;
```

```
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
    bedtime[args[0]->pr_addr] = 0;
}

END
{
    printa("%d/%d sleeping on %d/%d:\n%@d\n", @);
}
```

Wenn Sie dieses geänderte Skript einige Sekunden lang ausführen, erhalten Sie eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./firebird.d
^C

 100459/1 sleeping on 100459/13:

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

100459/13 sleeping on 100459/1:

           value  -------------- Distribution ------------ count
        16777216 |                                         0
        33554432 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1
        67108864 |                                         0

100459/1 sleeping on 100459/2:

           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@                                     5
           65536 |@                                        2
          131072 |@@@@@                                    6
          262144 |                                         1
          524288 |@                                        2
         1048576 |                                         0
         2097152 |@@                                       3
         4194304 |@@@@                                     5
         8388608 |@@@@@@@@                                 9
        16777216 |@@@@@                                    6
        33554432 |@@                                       3
```

```
        67108864 |                                         0

100459/1 sleeping on 100459/5:

           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@@                                    12
           65536 |@@                                       5
          131072 |@@@@@@                                   15
          262144 |                                         1
          524288 |                                         1
         1048576 |                                         2
         2097152 |@                                        4
         4194304 |@@@@@                                    13
         8388608 |@@@                                      8
        16777216 |@@@@@                                    13
        33554432 |@@                                       6
        67108864 |@@                                       5
       134217728 |@                                        4
       268435456 |                                         0
       536870912 |                                         1
      1073741824 |                                         0

100459/2 sleeping on 100459/1:

           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@@@@@@@@@@@@@@                           11
           65536 |                                         0
          131072 |@@                                       2
          262144 |                                         0
          524288 |                                         0
         1048576 |@@@@                                     3
         2097152 |@                                        1
         4194304 |@@                                       2
         8388608 |@@                                       2
        16777216 |@                                        1
        33554432 |@@@@@@                                   5
        67108864 |                                         0
       134217728 |                                         0
       268435456 |                                         0
       536870912 |@                                        1
      1073741824 |@                                        1
      2147483648 |@                                        1
      4294967296 |                                         0

100459/5 sleeping on 100459/1:
```

```
           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |                                         1
           65536 |                                         2
          131072 |                                         4
          262144 |                                         7
          524288 |                                         1
         1048576 |                                         5
         2097152 |                                         10
         4194304 |@@@@@@                                   77
         8388608 |@@@@@@@@@@@@@@@@@@@@@@@@                 270
        16777216 |@@@                                      43
        33554432 |@                                        20
        67108864 |@                                        14
       134217728 |                                         5
       268435456 |                                         2
       536870912 |                                         1
      1073741824 |                                         0
```

Auch die Leistung von Door-Servern wie zum Beispiel des Cache-Dämons des Namensdienstes kann, wie das folgende Beispiel zeigt, mit den Prüfpunkten `sleep` und `wakeup` beobachtet werden:

```
sched:::sleep
/curlwpsinfo->pr_stype == SOBJ_SHUTTLE/
{
    bedtime[curlwpsinfo->pr_addr] = timestamp;
}

sched:::wakeup
/execname == "nscd" && bedtime[args[0]->pr_addr]/
{
    @[stringof(curpsinfo->pr_fname), stringof(args[1]->pr_fname)] =
        quantize(timestamp - bedtime[args[0]->pr_addr]);
    bedtime[args[0]->pr_addr] = 0;
}

sched:::wakeup
/bedtime[args[0]->pr_addr]/
{
    bedtime[args[0]->pr_addr] = 0;
}
```

Das Ende der Ausgabe, die Sie erhalten, wenn Sie dieses Skript auf einem großen Mail-Server ausführen, sieht etwa wie folgt aus:

```
imapd
           value  -------------- Distribution ------------ count
           16384 |                                         0
```

```
          32768 |                                         2
          65536 |@@@@@@@@@@@@@@@@@@                       57
         131072 |@@@@@@@@@@@                              37
         262144 |                                         3
         524288 |@@@                                      11
        1048576 |@@@                                      10
        2097152 |@@                                       9
        4194304 |                                         1
        8388608 |                                         0

mountd
           value  -------------- Distribution ------------ count
           65536 |                                         0
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@      49
          262144 |@@@                                      6
          524288 |                                         1
         1048576 |                                         0
         2097152 |                                         0
         4194304 |@@@@                                     7
         8388608 |@                                        3
        16777216 |                                         0

sendmail
           value  -------------- Distribution ------------ count
           16384 |                                         0
           32768 |@                                        18
           65536 |@@@@@@@@@@@@@@@@@@@                      205
          131072 |@@@@@@@@@@@@@@                           154
          262144 |@                                        23
          524288 |                                         5
         1048576 |@@@@                                     50
         2097152 |                                         7
         4194304 |                                         5
         8388608 |                                         2
        16777216 |                                         0

automountd
           value  -------------- Distribution ------------ count
           32768 |                                         0
           65536 |@@@@@@@@@@                               22
          131072 |@@@@@@@@@@@@@@@@@@@@@@@@@                51
          262144 |@@                                       6
          524288 |                                         1
         1048576 |                                         0
         2097152 |                                         2
         4194304 |                                         2
         8388608 |                                         1
        16777216 |                                         1
```

```
        33554432 |                                       1
        67108864 |                                       0
       134217728 |                                       0
       268435456 |                                       1
       536870912 |                                       0
```

Möglicherweise fallen Ihnen die ungewöhnlichen Datenpunkte für `automountd` oder der dauerhafte Datenpunkt bei über einer Millisekunde für `sendmail` als interessant auf. Mithilfe von zusätzlichen Prädikaten, die Sie in das obige Skript einbauen, können Sie die Ursachen für jedes außergewöhnliche oder anomale Ergebnis weiter einkreisen.

`preempt,remain-cpu`

Da Solaris ein präemptives System ist, werden Threads mit niedriger Priorität unterbrochen, indem Threads mit höherer Priorität CPU-Zeit zugeteilt wird. Diese Vorrangunterbrechung kann zu einer bedeutenden Latenz des Threads mit der niedrigeren Priorität führen. Deshalb ist es mitunter hilfreich, zu wissen, welche Threads durch welche anderen unterbrochen werden. Das folgende Beispiel zeigt, wie sich diese Informationen mithilfe der Prüfpunkte `preempt` und `remain-cpu` anzeigen lassen:

```
#pragma D option quiet

sched:::preempt
{
    self->preempt = 1;
}

sched:::remain-cpu
/self->preempt/
{
    self->preempt = 0;
}

sched:::off-cpu
/self->preempt/
{
    /*
     * If we were told to preempt ourselves, see who we ended up giving
     * the CPU to.
     */
    @[stringof(args[1]->pr_fname), args[0]->pr_pri, execname,
        curlwpsinfo->pr_pri] = count();
    self->preempt = 0;
}

END
```

```
{
    printf("%30s %3s %30s %3s %5s\n", "PREEMPTOR", "PRI",
        "PREEMPTED", "PRI", "#");
    printa("%30s %3d %30s %3d %5@d\n", @);
}
```

Wenn Sie das obige Skript einige Sekunden lang auf einem Desktop-System ausführen, erhalten Sie eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./whopreempt.d
^C
                      PREEMPTOR PRI                      PREEMPTED PRI     #
                          sched  60                           Xsun  53     1
                          xterm  59                           Xsun  53     1
                MozillaFirebird  57                           Xsun  53     1
                          mpstat 100                         fvwm2  59     1
                          sched  99                MozillaFirebird  57     1
                          sched  60                          dtrace  30     1
                          mpstat 100                          Xsun  59     2
                          sched  60                           Xsun  54     2
                          sched  99                          sched  60     2
                          fvwm2  59                           Xsun  44     2
                          sched  99                           Xsun  44     2
                          sched  60                          xterm  59     2
                          sched  99                           Xsun  53     2
                          sched  99                           Xsun  54     3
                          sched  60                          fvwm2  59     3
                          sched  60                           Xsun  59     3
                          sched  99                           Xsun  59     4
                          fvwm2  59                           Xsun  54     8
                          fvwm2  59                           Xsun  53     9
                           Xsun  59                MozillaFirebird  57    10
                          sched  60                MozillaFirebird  57    14
                MozillaFirebird  57                           Xsun  44    16
                MozillaFirebird  57                           Xsun  54    18
```

change-pri

Die Vorrangunterbrechung basiert auf Prioritäten. Aus diesem Grund kann sich auch die Beobachtung der Veränderung von Prioritäten über einen bestimmten Zeitraum als interessant erweisen. Im nächsten Beispiel werden diese Informationen mithilfe des Prüfpunkts `change-pri` angezeigt:

```
sched:::change-pri
{
    @[stringof(args[0]->pr_clname)] =
```

```
        lquantize(args[2] - args[0]->pr_pri, -50, 50, 5);
}
```

Das Beispielskript erfasst den Grad, auf den die Priorität angehoben oder herabgesetzt wird, und aggregiert die Daten nach Scheduling-Klasse. Die Ausführung des obigen Skripts erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./pri.d
dtrace: script ’./pri.d’ matched 10 probes
^C
 IA
           value  -------------- Distribution ------------ count
           < -50 |                                         20
             -50 |@                                        38
             -45 |                                         4
             -40 |                                         13
             -35 |                                         12
             -30 |                                         18
             -25 |                                         18
             -20 |                                         23
             -15 |                                         6
             -10 |@@@@@@@@                                 201
              -5 |@@@@@@                                   160
               0 |@@@@@                                    138
               5 |@                                        47
              10 |@@                                       66
              15 |@                                        36
              20 |@                                        26
              25 |@                                        28
              30 |                                         18
              35 |                                         22
              40 |                                         8
              45 |                                         11
           >= 50 |@                                        34

  TS
           value  -------------- Distribution ------------ count
             -15 |                                         0
             -10 |@                                        1
              -5 |@@@@@@@@@@@@                             7
               0 |@@@@@@@@@@@@@@@@@@@@                     12
               5 |                                         0
              10 |@@@@@                                    3
              15 |                                         0
```

Die Ausgabe zeigt die Manipulation der Priorität für die interaktive (IA) Scheduling-Klasse. Anstelle der *Manipulation* der Priorität möchten Sie aber womöglich die Prioritäts*werte* eines bestimmten Prozesses und Threads im Verlauf der Zeit feststellen. Im nächsten Skript werden diese Informationen mithilfe des Prüfpunkts `change-pri` angezeigt:

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
}

sched:::change-pri
/args[1]->pr_pid == $1 && args[0]->pr_lwpid == $2/
{
    printf("%d %d\n", timestamp - start, args[2]);
}

tick-1sec
/++n == 5/
{
    exit(0);
}
```

Um die Änderung der Priorität im Verlauf der Zeit zu sehen, geben Sie den folgenden Befehl in ein Fenster ein:

```
$ echo $$
139208
$ while true ; do let i=0 ; done
```

Führen Sie in einem anderen Fenster das Skript aus und leiten Sie die Ausgabe in eine Datei um:

```
# dtrace -s ./pritime.d 139208 1 > /tmp/pritime.out
#
```

Die dabei generierte Datei `/tmp/pritime.out` kann als Eingabe für Plotting-Software verwendet werden und ermöglicht so die grafische Anzeige der Prioritätsänderung im Verlauf der Zeit. `gnuplot` ist auf der Solaris Freeware Companion-CD als frei verfügbare Plotting-Software enthalten. `gnuplot` wird standardmäßig unter `/opt/sfw/bin` installiert.

tick

In Solaris kommt eine *tick-basierte CPU-Abrechnung* zum Einsatz. Dabei wird in einem festgelegten Intervall ein Systemuhr-Interrupt ausgelöst und den zum Zeitpunkt des Ticks laufenden Threads und Prozessen CPU-Zeit zugewiesen. Das nächste Beispiel veranschaulicht, wie sich diese Zuweisung mit dem Prüfpunkt `tick` beobachten lässt:

```
# dtrace -n sched:::tick'{@[stringof(args[1]->pr_fname)] = count()}'
^C
  arch                                                              1
```

```
  sh                                                1
  sed                                               1
  echo                                              1
  ls                                                1
  FvwmAuto                                          1
  pwd                                               1
  awk                                               2
  basename                                          2
  expr                                              2
  resize                                            2
  tput                                              2
  uname                                             2
  fsflush                                           2
  dirname                                           4
  vim                                               9
  fvwm2                                            10
  ksh                                              19
  xterm                                            21
  Xsun                                             93
  MozillaFirebird                                 260
```

Die Systemuhrfrequenz ist von Betriebssystem zu Betriebssystem verschieden, bewegt sich aber im Allgemeinen zwischen 25 und 1024 Hertz. Unter Solaris ist dieser Systemtakt anpassbar, beträgt aber standardmäßig 100 Hertz.

Der Prüfpunkt `tick` wird nur ausgelöst, wenn die Systemuhr einen ausführbaren Thread entdeckt. Damit der Systemtakt anhand des Prüfpunkts `tick` beobachtet werden kann, muss ein stets ausführbarer Thread vorliegen. Erzeugen Sie in einem Fenster wie folgt eine Shell-Schleife:

```
$ while true ; do let i=0 ; done
```

Führen Sie in einem anderen Fenster das folgende Skript aus:

```
uint64_t last[int];

sched:::tick
/last[cpu]/
{
    @[cpu] = min(timestamp - last[cpu]);
}

sched:::tick
{
    last[cpu] = timestamp;
}
```

```
# dtrace -s ./ticktime.d
dtrace: script ’./ticktime.d’ matched 2 probes
^C

  0          9883789
```

Das geringste Intervall beträgt 9,8 Millisekunden, was darauf hindeutet, dass der Standardtakt 10 Millisekunden (100 Hertz) beträgt. Der beobachtete Mindestwert liegt aufgrund von Jitter ein wenig unter 10 Millisekunden.

Ein Mangel der tick-basierten Abrechnung besteht darin, dass die die Abrechnung durchführende Systemuhr häufig auch für das Dispatchen zeitorientierter Scheduling-Aktivitäten verantwortlich ist. Folglich rechnet das System für Threads, die mit jedem Uhr-Tick (d. h. alle 10 Millisekunden) einen gewissen Umfang an Arbeit leisten sollen, je nachdem, ob die Abrechnung vor oder nach dem Dispatchen der zeitorientierten Scheduling-Aktivität erfolgt, entweder zu viel oder zu wenig Zeit ab. Unter Solaris erfolgt die Abrechnung vor dem zeitorientierten Dispatchen. Daraus folgt, dass das System Threads, die in einem regelmäßigen Intervall ausgeführt werden, zu niedrig abrechnet. Wenn solche Threads kürzer als die Dauer eines Uhr-Tick-Intervalls ausfallen, können sie sich erfolgreich hinter dem Uhr-Tick „verbergen“. Das folgende Beispiel zeigt, in welchem Umfang derartige Threads im System vorkommen:

```
sched:::tick,
sched:::enqueue
{
    @[probename] = lquantize((timestamp / 1000000) % 10, 0, 10);
}
```

Die Ausgabe des Beispielskripts zeigt zwei Verteilungen des Versatzes in Millisekunden innerhalb eines Intervalls von zehn Millisekunden - eine für den Prüfpunkt `tick` und eine weitere für `enqueue`:

```
# dtrace -s ./tick.d
dtrace: script ’./tick.d’ matched 4 probes
^C
  tick
           value  ------------- Distribution ------------ count
               6 |                                         0
               7 |@                                        3
               8 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   79
               9 |                                         0

  enqueue
           value  ------------- Distribution ------------ count
             < 0 |                                         0
               0 |@@                                       267
               1 |@@                                       300
```

```
           2 |@@                                       259
           3 |@@                                       291
           4 |@@@                                      360
           5 |@@                                       305
           6 |@@                                       295
           7 |@@@@                                     522
           8 |@@@@@@@@@@@@@                           1315
           9 |@@@                                      337
```

Das Ausgabehistogramm `tick` zeigt, dass der Uhr-Tick mit einem Versatz von 8 Millisekunden ausgelöst wird. Wenn das Scheduling unabhängig von dem Uhr-Tick erfolgen würde, wäre die Ausgabe für `enqueue` gleichmäßig über das Intervall von zehn Millisekunden verteilt. Wir sehen in der Ausgabe jedoch an demselben Versatz von 8 Millisekunden eine Spitze, die darauf hinweist, dass für einige Threads im System *sehr wohl* ein zeitorientiertes Scheduling stattfindet.

Stabilität

Der Provider `sched` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Argumente	Evolving	Evolving	ISA

◆ ◆ ◆ KAPITEL 27

Der Provider io

Der Provider io stellt Prüfpunkte für die Festplatten-E/A zur Verfügung. Mithilfe des Providers iolässt sich einem mit E/A-Überwachungstools wie beispielsweise iostat(1M) beobachteten Verhalten schnell auf den Grund gehen. So können Sie mit dem Provider io die Ein- und Ausgabe beispielsweise nach Gerät, nach E/A-Typ, nach E/A-Größe, nach Prozess, Anwendungsnamen, Dateinamen oder nach Dateiversatz untersuchen.

Prüfpunkte

Die io-Prüfpunkte sind in Tabelle 27–1 beschrieben.

TABELLE 27–1 io-Prüfpunkte

Prüfpunkt	Beschreibung
start	Prüfpunkt, der ausgelöst wird, kurz bevor eine E/A-Anforderung an ein Peripheriegerät oder einen NFS-Server ausgegeben wird. Auf die Struktur bufinfo_t der E/A-Anforderung zeigt args[0]. Auf devinfo_t des Geräts, an das die E/A-Anforderung gestellt wird, zeigt args[1]. Auf die Struktur fileinfo_t der Datei, die mit der E/A-Anforderung übereinstimmt, zeigt args[2]. Beachten Sie, dass die Verfügbarkeit von Dateiinformationen davon abhängig ist, welches Dateisystem die E/A-Anforderung stellt. Weitere Informationen finden Sie unter „fileinfo_t“ auf Seite 319.
done	Prüfpunkt, der nach der Erfüllung einer E/A-Anforderung ausgelöst wird. Auf die Struktur bufinfo_t der E/A-Anforderung zeigt args[0]. Der Prüfpunkt done wird nach Abschluss der E/A ausgelöst, aber noch bevor im Puffer die Abschlussverarbeitung durchgeführt wird. B_DONE ist also zum Zeitpunkt der Auslösung des Prüfpunkts done *nicht* in b_flags gesetzt. Auf devinfo_t des Geräts, an das die E/A-Anforderung gesendet wurde, zeigt args[1]. Auf die Struktur fileinfo_t der Datei, die mit der E/A-Anforderung übereinstimmt, zeigt args[2].

TABELLE 27–1 io-Prüfpunkte *(Fortsetzung)*

Prüfpunkt	Beschreibung
wait-start	Prüfpunkt, der ausgelöst wird, unmittelbar bevor ein Thread beginnt, auf die Durchführung einer gegebenen E/A-Anforderung zu warten. args[0] zeigt auf die Struktur buf(9S) der E/A-Anforderung, auf die der Thread wartet. Auf devinfo_t des Geräts, an das die E/A-Anforderung gesendet wurde, zeigt args[1] . Auf die Struktur fileinfo_t der Datei, die mit der E/A-Anforderung übereinstimmt, zeigt args[2]. Einige Zeit, nachdem der Prüfpunkt wait-start ausgelöst wurde, wird der Prüfpunkt wait-done in demselben Thread ausgelöst.
wait-done	Prüfpunkt, der ausgelöst wird, wenn ein Thread aufhört, auf die Durchführung einer E/A-Anforderung zu warten. Auf die Struktur bufinfo_t, die der E/A-Anforderung entspricht, auf die der Thread wartet, zeigt args[0]. Auf devinfo_t des Geräts, an das die E/A-Anforderung gesendet wurde, zeigt args[1]. Auf die Struktur fileinfo_t der Datei, die mit der E/A-Anforderung übereinstimmt, zeigt args[2]. Der Prüfpunkt wait-done wird erst nach der Auslösung des Prüfpunkts wait-start in demselben Thread ausgelöst.

Beachten Sie, dass die io-Prüfpunkte für sämtliche E/A-Anforderungen an Peripheriegeräte und sämtliche Lese- und Schreibzugriffsanforderungen für Dateien an einen NFS-Server ausgelöst werden. Anforderungen von Metadaten von einem NFS-Server lösen io-Prüfpunkte aufgrund einer readdir(3C)-Anforderung beispielsweise *nicht* aus.

Argumente

Die Argumenttypen für io-Prüfpunkte sind in Tabelle 27–2 aufgeführt. In Tabelle 27–1 finden Sie eine Beschreibung der Argumente.

TABELLE 27–2 Argumente für io-Prüfpunkte

Prüfpunkt	args[0]	args[1]	args[2]
start	struct buf *	devinfo_t *	fileinfo_t *
done	struct buf *	devinfo_t *	fileinfo_t *
wait-start	struct buf *	devinfo_t *	fileinfo_t *
wait-done	struct buf *	devinfo_t *	fileinfo_t *

Jeder io-Prüfpunkt besitzt Argumente in Form eines Zeigers auf eine buf(9S)-Struktur, eines Zeigers auf eine devinfo_t- und eines Zeigers auf eine fileinfo_t-Struktur. Dieser Abschnitt befasst sich ausführlich mit diesen Strukturen.

Die Struktur `bufinfo_t`

Die Struktur `bufinfo_t` ist die eine E/A-Anforderung beschreibende Abstraktion. Auf den Puffer für eine E/A-Anforderung wird mit `args[0]` in den Prüfpunkten `start`, `done`, `wait-start` und `wait-done` gezeigt. Die Definition der Struktur `bufinfo_t` lautet:

```
typedef struct bufinfo {
    int b_flags;                    /* flags */
    size_t b_bcount;                /* number of bytes */
    caddr_t b_addr;                 /* buffer address */
    uint64_t b_blkno;               /* expanded block # on device */
    uint64_t b_lblkno;              /* block # on device */
    size_t b_resid;                 /* # of bytes not transferred */
    size_t b_bufsize;               /* size of allocated buffer */
    caddr_t b_iodone;               /* I/O completion routine */
    dev_t b_edev;                   /* extended device */
 } bufinfo_t;
```

Die Komponente `b_flags` gibt den Status des E/A-Puffers an und besteht aus einem Bitwise-oder anderen Statuswerten. Tabelle 27–3 zeigt die gültigen Statuswerte.

TABELLE 27–3 Werte für `b_flags`

`B_DONE`	Zeigt an, dass die Datenübertragung abgeschlossen ist.
`B_ERROR`	Deutet auf einen E/A-Übertragungsfehler hin. Wird in Verbindung mit dem Feld `b_error` gesetzt.
`B_PAGEIO`	Gibt an, dass der Puffer für eine per Paging behandelte E/A-Anforderung verwendet wird. In der Beschreibung des Felds `b_addr` finden Sie weitere Informationen.
`B_PHYS`	Gibt an, dass der Puffer für eine physische (direkte) E/A an einen Benutzerdatenbereich verwendet wird.
`B_READ`	Gibt an, dass die Daten aus dem Peripheriegerät in den Hauptspeicher eingelesen werden müssen.
`B_WRITE`	Gibt an, dass die Daten aus dem Hauptspeicher an das Peripheriegerät übertragen werden müssen.
`B_ASYNC`	Die E/A-Anforderung ist asynchron. Auf sie wird nicht gewartet. Die Prüfpunkte `wait-start` und `wait-done` werden für asynchrone E/A-Anforderungen nicht ausgelöst. Beachten Sie, dass `B_ASYNC` bei manchen als asynchron vorgesehenen E/A nicht gesetzt ist: Das Subsystem für asynchrone E/A kann die asynchrone Anforderung unter Umständen durch einen separaten Arbeiter-Thread implementieren, der eine synchrone E/A-Operation durchführt.

Das Feld `b_bcount` gibt die Anzahl der im Rahmen der E/A-Anforderung zu übertragenden Byte an.

Das Feld `b_addr` ist die virtuelle Adresse der E/A-Anforderung, sofern `B_PAGEIO` nicht gesetzt ist. Bei der Adresse handelt es sich um eine virtuelle Kernel-Adresse, sofern `B_PHYS` nicht gesetzt ist. In diesem Fall handelt es sich um eine virtuelle Benutzer-Adresse. Wenn `B_PAGEIO` gesetzt ist, enthält das Feld `b_addr` private Kerneldaten. Es kann entweder genau eines der Flags `B_PHYS` und `B_PAGEIO` oder aber keines gesetzt sein.

Das Feld `b_lblkno` gibt an, auf welchen logischen Block auf dem Gerät zugegriffen werden muss. Die Zuordnung zwischen einem logischen und einem physischen Block (z. B. Zylinder oder Spur) ist durch das Gerät definiert.

Das Feld `b_resid` ist auf die Anzahl der aufgrund eines Fehlers nicht übertragenen Byte gesetzt.

Das Feld `b_bufsize` enthält die Größe des zugewiesenen Puffers.

Das Feld `b_iodone` gibt eine bestimmte Routine im Kernel an, die aufgerufen wird, wenn die E/A abgeschlossen ist.

Das Feld `b_error` kann einen im Fall eines E/A-Fehlers vom Treiber gelieferten Fehlercode enthalten. `b_error` wird in Verbindung mit dem `B_ERROR`-Bit in der Komponente `b_flags` gesetzt.

Das Feld `b_edev` enthält die Geräteklasse und Unternummer des Geräts, auf das zugegriffen wird. Die Verbraucher können die Geräteklasse und Unternummer mithilfe der D-Subroutinen `getmajor()` und `getminor()` aus dem Feld `b_edev` extrahieren.

devinfo_t

Die Struktur `devinfo_t` liefert Informationen über Geräte. Auf die Struktur `devinfo_t`, die dem Zielgerät einer E/A entspricht, wird mit `args[1]` in den Prüfpunkten `start`, `done`, `wait-start` und `wait-done` gezeigt. Die Komponenten von `devinfo_t` lauten:

```
typedef struct devinfo {
    int dev_major;                  /* major number */
    int dev_minor;                  /* minor number */
    int dev_instance;               /* instance number */
    string dev_name;                /* name of device */
    string dev_statname;            /* name of device + instance/minor */
    string dev_pathname;            /* pathname of device */
} devinfo_t;
```

Das Feld `dev_major` stellt die Geräteklasse des Geräts dar. Weitere Informationen finden Sie unter `getmajor(9F)`.

Das Feld `dev_minor` stellt die Unternummer des Geräts dar. Weitere Informationen finden Sie unter `getminor(9F)`.

Das Feld `dev_instance` gibt die Instanznummer des Geräts an. Die Instanz eines Geräts ist nicht mit der Unternummer identisch. Die Unternummer ist eine vom Gerätetreiber verwaltete Abstraktion. Bei der Instanznummer handelt es sich um eine Eigenschaft des Geräteknotens. Mit `prtconf(1M)` können die Instanznummern von Geräteknoten angezeigt werden.

Das Feld `dev_name` stellt den Namen des Gerätetreibers dar, der das Gerät verwaltet. Mit der Option `-D`für `prtconf(1M)` lassen sich die Gerätetreibernamen anzeigen.

Das Feld `dev_statname`gibt den Namen des Geräts an, wie er von `iostat(1M)` gemeldet wird. Dieser Name entspricht auch dem Namen einer von `kstat(1M)` ausgegebenen Kernelstatistik. Das Feld wird bereitgestellt, damit sich abweichende `iostat`- oder `kstat`-Ausgaben schnell mit der tatsächlichen E/A-Aktivität in Verbindung bringen lassen.

Das Feld `dev_pathname` gibt den vollständigen Pfad des Geräts an. Dieser Pfad kann `prtconf(1M)` als Argument übergeben werden, wenn ausführliche Geräteinformationen gewünscht sind. Die Komponenten des mit `dev_pathname` angegebenen Pfads geben den Geräteknoten, die Instanznummer und den Unterknoten wieder. Der Statistikname enthält jedoch nicht unbedingt alle drei Elemente. Bei einigen Geräten besteht der Statistikname aus dem Gerätenamen und der Instanznummer, bei anderen aus dem Gerätenamen und der Nummer des Unterknotens. Das bedeutet, dass zwei Geräte mit demselben `dev_statname`-Wert durchaus einen unterschiedlichen `dev_pathname`-Wert aufweisen können .

`fileinfo_t`

Die Struktur `fileinfo_t` liefert Informationen über Dateien. Auf die Datei, auf die sich eine E/A bezieht, wird mit `args[2]` in den Prüfpunkten `start`, `done`, `wait-start` und `wait-done` gezeigt. Ob Dateiinformationen vorliegen, hängt davon ab, ob das Dateisystem diese Informationen beim Dispatchen von E/A-Anforderungen bereitstellt. Einige Dateisysteme, insbesondere von anderen Herstellern, geben diese Informationen möglicherweise nicht an. Außerdem können E/A-Anforderungen aus Dateisystemen stammen, für die keine Dateiinformationen existieren. E/A-Operationen mit Dateisystem-Metadaten werden zum Beispiel mit keiner einzelnen Datei in Verbindung gebracht. Schließlich können einige hoch optimierte Dateisysteme E/A aus separaten Dateien in einer einzigen E/A-Anforderung zusammenfassen. In diesem Fall macht·das Dateisystem Dateiinformationen entweder für die Datei, die die meisten E/A-Operationen oder die Datei, die *einige* der E/A-Operationen durchführt, verfügbar. Alternativ dazu kann es sein, dass das Dateisystem in diesem Fall überhaupt keine Informationen zur Verfügung stellt.

Die Definition der Struktur `fileinfo_t` lautet:

```
typedef struct fileinfo {
    string fi_name;                 /* name (basename of fi_pathname) */
    string fi_dirname;              /* directory (dirname of fi_pathname) */
    string fi_pathname;             /* full pathname */
```

```
    offset_t fi_offset;             /* offset within file */
    string fi_fs;                   /* filesystem */
    string fi_mount;                /* mount point of file system */
} fileinfo_t;
```

Das Feld `fi_name` enthält den Namen der Datei, aber keine Verzeichniskomponenten. Wenn keine Dateiinformationen mit einer E/A in Verbindung gebracht werden können, wird das Feld `fi_name` auf die Zeichenkette `<none>` gesetzt. In seltenen Fällen ist der Pfadname für eine Datei unbekannt. Dann wird das Feld `fi_name` auf die Zeichenkette `<unknown>` gesetzt.

Das Feld `fi_dirname` enthält *nur* die Verzeichniskomponente des Dateinamens. Ebenso wie bei `fi_name` kann diese Zeichenkette auf `<none>` gesetzt sein, wenn keine Dateiinformationen vorliegen, oder auf `<unknown>`, wenn der Pfadname der Datei unbekannt ist.

Das Feld `fi_pathname` enthält den vollständigen Pfadnamen der Datei. Ebenso wie bei `fi_name` kann diese Zeichenkette auf `<none>` gesetzt sein, wenn keine Dateiinformationen vorliegen, oder auf `<unknown>`, wenn der Pfadname der Datei unbekannt ist.

Das Feld `fi_offset` enthält den Versatz innerhalb der Datei oder -1, wenn entweder keine Dateiinformationen vorliegen oder der Versatz auf andere Weise nicht vom Dateisystem angegeben wird.

Beispiele

Das folgende Beispielskript zeigt bei der Ausgabe jeder E/A einschlägige Informationen an:

```
#pragma D option quiet

BEGIN
{
    printf("%10s %58s %2s\n", "DEVICE", "FILE", "RW");
}

io:::start
{
    printf("%10s %58s %2s\n", args[1]->dev_statname,
        args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W");
}
```

Bei einem Kaltstart von Acrobat Reader auf einem x86-Laptop erhalten wir eine Ausgabe der Art:

```
# dtrace -s ./iosnoop.d
    DEVICE                                                  FILE RW
     cmdk0                                 /opt/Acrobat4/bin/acroread  R
     cmdk0                                 /opt/Acrobat4/bin/acroread  R
```

```
cmdk0                                                  <unknown>  R
cmdk0                           /opt/Acrobat4/Reader/AcroVersion  R
cmdk0                                                  <unknown>  R
cmdk0                                                  <unknown>  R
cmdk0                                                     <none>  R
cmdk0                                                  <unknown>  R
cmdk0                                                     <none>  R
cmdk0                 /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                 /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                 /usr/lib/locale/iso_8859_1/iso_8859_1.so.3  R
cmdk0                                                     <none>  R
cmdk0                                                  <unknown>  R
cmdk0                                                  <unknown>  R
cmdk0                                                  <unknown>  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                                     <none>  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                                  <unknown>  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0                                                     <none>  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0   /opt/Acrobat4/Reader/intelsolaris/lib/libreadcore.so.4.0  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0             /opt/Acrobat4/Reader/intelsolaris/bin/acroread  R
cmdk0                                                  <unknown>  R
cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R
cmdk0                                                     <none>  R
cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R
cmdk0        /opt/Acrobat4/Reader/intelsolaris/lib/libAGM.so.3.0  R
  ...
```

Die `<none>`-Einträge in der Ausgabe deuten darauf hin, dass sich die E/A auf keine der Daten in einer bestimmten Datei bezieht: Diese E/A-Operationen beziehen sich auf Metadaten in der einen oder anderen Form. Die `<unknown>`-Einträge in der Ausgabe bedeuten, dass der Pfadname der Datei unbekannt ist. Diese Situation ergibt sich relativ selten.

Das Beispielskript ließe sich durch einen assoziativen Vektor zum Erfassen der Dauer jeder E/A ein wenig verfeinern. Dies zeigt das nächste Beispiel:

```
#pragma D option quiet

BEGIN
{
    printf("%10s %58s %2s %7s\n", "DEVICE", "FILE", "RW", "MS");
}

io:::start
{
    start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
}

io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
    this->elapsed = timestamp - start[args[0]->b_edev, args[0]->b_blkno];
    printf("%10s %58s %2s %3d.%03d\n", args[1]->dev_statname,
        args[2]->fi_pathname, args[0]->b_flags & B_READ ? "R" : "W",
        this->elapsed / 10000000, (this->elapsed / 1000) % 1000);
    start[args[0]->b_edev, args[0]->b_blkno] = 0;
}
```

Das nächste Beispiel zeigt die Ausgabe des obigen Beispielskripts beim Hot-Plugging eines USB-Speichergeräts in ein ansonsten im Ruhezustand befindliches x86-Laptop-System:

```
# dtrace -s ./iotime.d
    DEVICE                                                  FILE RW      MS
     cmdk0                                  /kernel/drv/scsa2usb  R  24.781
     cmdk0                                  /kernel/drv/scsa2usb  R  25.208
     cmdk0                                     /var/adm/messages  W  25.981
     cmdk0                                  /kernel/drv/scsa2usb  R   5.448
     cmdk0                                                <none>  W   4.172
     cmdk0                                  /kernel/drv/scsa2usb  R   2.620
     cmdk0                                     /var/adm/messages  W   0.252
     cmdk0                                             <unknown>  R   3.213
     cmdk0                                                <none>  W   3.011
     cmdk0                                             <unknown>  R   2.197
     cmdk0                                     /var/adm/messages  W   2.680
     cmdk0                                                <none>  W   0.436
     cmdk0                                     /var/adm/messages  W   0.542
     cmdk0                                                <none>  W   0.339
     cmdk0                                     /var/adm/messages  W   0.414
     cmdk0                                                <none>  W   0.344
     cmdk0                                     /var/adm/messages  W   0.361
     cmdk0                                                <none>  W   0.315
```

```
cmdk0                                        /var/adm/messages  W   0.421
cmdk0                                                  <none>  W   0.349
cmdk0                                                  <none>  R   1.524
cmdk0                                                <unknown>  R   3.648
cmdk0                                     /usr/lib/librcm.so.1  R   2.553
cmdk0                                     /usr/lib/librcm.so.1  R   1.332
cmdk0                                     /usr/lib/librcm.so.1  R   0.222
cmdk0                                     /usr/lib/librcm.so.1  R   0.228
cmdk0                                     /usr/lib/librcm.so.1  R   0.927
cmdk0                                                  <none>  R   1.189
  ...
cmdk0                               /usr/lib/devfsadm/linkmod  R   1.110
cmdk0         /usr/lib/devfsadm/linkmod/SUNW_audio_link.so  R   1.763
cmdk0         /usr/lib/devfsadm/linkmod/SUNW_audio_link.so  R   0.161
cmdk0           /usr/lib/devfsadm/linkmod/SUNW_cfg_link.so  R   0.819
cmdk0           /usr/lib/devfsadm/linkmod/SUNW_cfg_link.so  R   0.168
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_disk_link.so  R   0.886
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_disk_link.so  R   0.185
cmdk0        /usr/lib/devfsadm/linkmod/SUNW_fssnap_link.so  R   0.778
cmdk0        /usr/lib/devfsadm/linkmod/SUNW_fssnap_link.so  R   0.166
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_lofi_link.so  R   1.634
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_lofi_link.so  R   0.163
cmdk0            /usr/lib/devfsadm/linkmod/SUNW_md_link.so  R   0.477
cmdk0            /usr/lib/devfsadm/linkmod/SUNW_md_link.so  R   0.161
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.198
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.168
cmdk0          /usr/lib/devfsadm/linkmod/SUNW_misc_link.so  R   0.247
cmdk0     /usr/lib/devfsadm/linkmod/SUNW_misc_link_i386.so  R   1.735
  ...
```

Diese Ausgabe lässt verschiedene Folgerungen über die Mechanik des Systems zu. Beachten Sie erstens, die lange Durchführungsdauer der ersten E/A-Operationen, die je 25 Millisekunden betrug. Diese Dauer kann darauf zurückzuführen sein, dass das Gerät `cmdk0` dem Power-Management auf dem Laptop unterzogen wurde. Zweitens fällt die E/A durch das Laden des Treibers `scsa2usb`(7D) für die Behandlung des USB-Massenspeichergeräts auf. Drittens sind die Schreibzugriffe auf `/var/adm/messages` bei Meldung des Geräts zu beachten. Beachten Sie abschließend die Leszugriffe der Geräte-Link-Generatoren (die auf `link.so` endenden Dateien), die sich wahrscheinlich auf das neue Gerät beziehen.

Der Provider `io` ermöglicht eine tief greifende Interpretation der Ausgabe von `iostat`(1M). Nehmen wir an, es werde eine `iostat`-Ausgabe wie in folgendem Beispiel beobachtet:

```
extended device statistics
device       r/s    w/s   kr/s   kw/s wait actv  svc_t  %w  %b
cmdk0        8.0    0.0  399.8    0.0  0.0  0.0    0.8   0   1
sd0          0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
sd2          0.0  109.0    0.0  435.9  0.0  1.0    8.9   0  97
nfs1         0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
```

```
nfs2         0.0    0.0    0.0    0.0  0.0  0.0    0.0   0   0
```

Mit dem Skript `iotime.d` können wir diese E/A-Operationen, wie das nächste Beispiel zeigt, im Moment ihres Auftretens anzeigen:

```
  DEVICE                                          FILE RW      MS
     sd2                              /mnt/archives.tar  W   0.856
     sd2                              /mnt/archives.tar  W   0.729
     sd2                              /mnt/archives.tar  W   0.890
     sd2                              /mnt/archives.tar  W   0.759
     sd2                              /mnt/archives.tar  W   0.884
     sd2                              /mnt/archives.tar  W   0.746
     sd2                              /mnt/archives.tar  W   0.891
     sd2                              /mnt/archives.tar  W   0.760
     sd2                              /mnt/archives.tar  W   0.889
   cmdk0                 /export/archives/archives.tar  R   0.827
     sd2                              /mnt/archives.tar  W   0.537
     sd2                              /mnt/archives.tar  W   0.887
     sd2                              /mnt/archives.tar  W   0.763
     sd2                              /mnt/archives.tar  W   0.878
     sd2                              /mnt/archives.tar  W   0.751
     sd2                              /mnt/archives.tar  W   0.884
     sd2                              /mnt/archives.tar  W   0.760
     sd2                              /mnt/archives.tar  W   3.994
     sd2                              /mnt/archives.tar  W   0.653
     sd2                              /mnt/archives.tar  W   0.896
     sd2                              /mnt/archives.tar  W   0.975
     sd2                              /mnt/archives.tar  W   1.405
     sd2                              /mnt/archives.tar  W   0.724
     sd2                              /mnt/archives.tar  W   1.841
   cmdk0                 /export/archives/archives.tar  R   0.549
     sd2                              /mnt/archives.tar  W   0.543
     sd2                              /mnt/archives.tar  W   0.863
     sd2                              /mnt/archives.tar  W   0.734
     sd2                              /mnt/archives.tar  W   0.859
     sd2                              /mnt/archives.tar  W   0.754
     sd2                              /mnt/archives.tar  W   0.914
     sd2                              /mnt/archives.tar  W   0.751
     sd2                              /mnt/archives.tar  W   0.902
     sd2                              /mnt/archives.tar  W   0.735
     sd2                              /mnt/archives.tar  W   0.908
     sd2                              /mnt/archives.tar  W   0.753
```

Diese Ausgabe lässt die Vermutung zu, dass die Datei `archives.tar` von `cmdk0` eingelesen (in `/export/archives`) und auf das Gerät `sd2` (in `/mnt`) geschrieben wird. Das Vorhandensein von zwei Dateien mit dem Namen `archives.tar`, auf die separat, aber gleichzeitig zugegriffen wird, wirkt unwahrscheinlich. Um dem weiter auf den Grund zu gehen, können Sie, wie das nächste Beispiel zeigt, Gerät, Anwendung, Prozess-ID und übertragene Byte aggregieren:

```
#pragma D option quiet

io:::start
{
    @[args[1]->dev_statname, execname, pid] = sum(args[0]->b_bcount);
}

END
{
    printf("%10s %20s %10s %15s\n", "DEVICE", "APP", "PID", "BYTES");
    printa("%10s %20s %10d %15@d\n", @);
}
```

Wenn Sie dieses Skript einige Sekunden lang ausführen, erhalten Sie eine Ausgabe wie in folgendem Beispiel:

```
# dtrace -s ./whoio.d
^C
    DEVICE                  APP        PID           BYTES
     cmdk0                   cp        790         1515520
       sd2                   cp        790         1527808
```

Die Ausgabe zeigt, dass es sich bei dieser Aktivität *tatsächlich* um das Kopieren der Datei `archives.tar` von einem Gerät auf ein anderes handelt. Daraus folgt jedoch eine weitere nahe liegende Frage: Ist eines der Geräte schneller als das andere? Welches Gerät beschränkt den Kopiervorgang? Um diese Fragen zu beantworten, müssen wir anstelle der pro Sekunde übertragenen Byte den effektiven Datendurchsatz jedes Geräts in Erfahrung bringen. Dabei hilft uns das nächste Beispielskript:

```
#pragma D option quiet

io:::start
{
    start[args[0]->b_edev, args[0]->b_blkno] = timestamp;
}

io:::done
/start[args[0]->b_edev, args[0]->b_blkno]/
{
    /*
     * We want to get an idea of our throughput to this device in KB/sec.
     * What we have, however, is nanoseconds and bytes.  That is we want
     * to calculate:
     *
     *                        bytes / 1024
     *                  ------------------------
     *                  nanoseconds / 1000000000
     *
```

```
     * But we can't calculate this using integer arithmetic without losing
     * precision (the denomenator, for one, is between 0 and 1 for nearly
     * all I/Os).  So we restate the fraction, and cancel:
     *
     *     bytes      1000000000         bytes        976562
     *   --------- * -------------  =  --------- * -------------
     *      1024      nanoseconds          1        nanoseconds
     *
     * This is easy to calculate using integer arithmetic; this is what
     * we do below.
     */
    this->elapsed = timestamp - start[args[0]->b_edev, args[0]->b_blkno];
    @[args[1]->dev_statname, args[1]->dev_pathname] =
        quantize((args[0]->b_bcount * 976562) / this->elapsed);
    start[args[0]->b_edev, args[0]->b_blkno] = 0;
}

END
{
    printa("  %s (%s)\n%@d\n", @);
}
```

Wenn Sie das Beispielskript einige Sekunden lang ausführen, erhalten Sie die folgende Ausgabe:

```
  sd2 (/devices/pci@0,0/pci1179,1@1d/storage@2/disk@0,0:r)

           value  ------------- Distribution ------------- count
              32 |                                         0
              64 |                                         3
             128 |                                         1
             256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  2257
             512 |                                         1
            1024 |                                         0

  cmdk0 (/devices/pci@0,0/pci-ide@1f,1/ide@0/cmdk@0,0:a)

           value  ------------- Distribution ------------- count
             128 |                                         0
             256 |                                         1
             512 |                                         0
            1024 |                                         2
            2048 |                                         0
            4096 |                                         2
            8192 |@@@@@@@@@@@@@@@@@@@                      172
           16384 |@@@@@                                    52
           32768 |@@@@@@@@@@@                              108
           65536 |@@@                                      34
          131072 |                                         0
```

Die Ausgabe zeigt, dass `sd2` eindeutig das beschränkende Gerät ist. Der Durchsatz von `sd2` liegt zwischen 256K/s und 512K/s, während `cmdk0` eine E/A-Leistung von 8 MB/s bis zu über 64 MB/s bietet. Das Skript gibt sowohl den Namen, wie er in `iostat` angezeigt wird, als auch den vollständigen Pfadnamen des Geräts aus. Wenn Sie mehr über das Gerät herausfinden möchten, könnten Sie wie folgt den Gerätepfad zu `prtconf` angeben:

```
# prtconf -v /devices/pci@0,0/pci1179,1@1d/storage@2/disk@0,0
disk, instance #2 (driver name: sd)
    Driver properties:
        name='lba-access-ok' type=boolean dev=(29,128)
        name='removable-media' type=boolean dev=none
        name='pm-components' type=string items=3 dev=none
            value='NAME=spindle-motor' + '0=off' + '1=on'
        name='pm-hardware-state' type=string items=1 dev=none
            value='needs-suspend-resume'
        name='ddi-failfast-supported' type=boolean dev=none
        name='ddi-kernel-ioctl' type=boolean dev=none
    Hardware properties:
        name='inquiry-revision-id' type=string items=1
            value='1.04'
        name='inquiry-product-id' type=string items=1
            value='STORAGE DEVICE'
        name='inquiry-vendor-id' type=string items=1
            value='Generic'
        name='inquiry-device-type' type=int items=1
            value=00000000
        name='usb' type=boolean
        name='compatible' type=string items=1
            value='sd'
        name='lun' type=int items=1
            value=00000000
        name='target' type=int items=1
            value=00000000
```

Die hervorgehobenen Bezeichnungen deuten darauf hin, dass es sich um einen austauschbaren USB-Datenträger handelt.

In den Beispielen dieses Abschnitts wurden alle E/A-Anforderungen untersucht. Möglicherweise interessiert Sie aber nur eine bestimmte Anforderungsart. Das folgende Beispiel protokolliert die Verzeichnisse, in denen Schreibzugriffe stattfinden, sowie die Anwendungen, die diese Schreibzugriffe durchführen.

```
#pragma D option quiet

io:::start
/args[0]->b_flags & B_WRITE/
{
    @[execname, args[2]->fi_dirname] = count();
```

```
}

END
{
    printf("%20s %51s %5s\n", "WHO", "WHERE", "COUNT");
    printa("%20s %51s %5@d\n", @);
}
```

Wenn Sie dieses Beispielskript eine Weile auf einem belegten Desktop ausführen, erhalten Sie, wie die nächste Beispielausgabe zeigt, einige interessante Ergebnisse:

```
# dtrace -s ./whowrite.d
^C
              WHO                                               WHERE COUNT
               su                                           /var/adm     1
          fsflush                                               /etc     1
          fsflush                                                  /     1
          fsflush                                           /var/log     1
          fsflush                                    /export/bmc/lisa     1
              esd   /export/bmc/.phoenix/default/78cxczuy.slt/Cache     1
          fsflush                             /export/bmc/.phoenix     1
              esd         /export/bmc/.phoenix/default/78cxczuy.slt     1
               vi                                           /var/tmp     2
               vi                                               /etc     2
              cat                                             <none>     2
             bash                                                  /     2
               vi                                             <none>     3
            xterm                                           /var/adm     3
          fsflush                                         /export/bmc     7
  MozillaFirebird                                             <none>     8
              vim                                         /export/bmc     9
  MozillaFirebird                                         /export/bmc    10
          fsflush                                            /var/adm    11
         devfsadm                                                /dev    14
              ksh                                             <none>    71
              ksh                                         /export/bmc    71
          fsflush         /export/bmc/.phoenix/default/78cxczuy.slt   119
  MozillaFirebird         /export/bmc/.phoenix/default/78cxczuy.slt   119
          fsflush                                             <none>   211
  MozillaFirebird   /export/bmc/.phoenix/default/78cxczuy.slt/Cache   591
          fsflush   /export/bmc/.phoenix/default/78cxczuy.slt/Cache   666
            sched                                             <none>  2385
```

Wie aus der Ausgabe hervorgeht, beziehen sich nahezu alle Schreibvorgänge auf den Cache für Mozilla Firebird. Die mit <none> gekennzeichneten Schreibzugriffe gehen wahrscheinlich auf das UFS-Protokoll zurück und wurden ihrerseits durch andere Schreibvorgänge im Dateisystem ausgelöst. Ausführliche Informationen zur Protokollierung finden Sie unter `ufs`(7FS) Dieses Beispiel demonstriert, wie mithilfe des Providers `io` Probleme auf wesentlich

höheren Softwareebenen entdeckt werden können. In diesem Fall hat das Skript ein Konfigurationsproblem aufgespürt: Der Webbrowser würde sehr viel weniger E/A verursachen (sehr wahrscheinlich sogar überhaupt keine), wenn sich sein Cache in einem Verzeichnis in einem `tmpfs`(7FS)-Dateisystem befände.

In den bisherigen Beispielen kamen nur die Prüfpunkte `start` und `done` zum Einsatz. Mithilfe der Prüfpunkte `wait-start` und `wait-done` können wir der Frage auf den Grund gehen, weshalb und wie lange Anwendungen in Erwartung von E/A blockieren. Im folgenden Beispielskript werden sowohl `io`- als auch `sched`-Prüfpunkte (siehe Kapitel 26, „Der Provider `sched`“) verwendet und die CPU-Zeit im Vergleich zur E/A-Wartezeit für die StarOffice-Software abgeleitet:

```
#pragma D option quiet

sched:::on-cpu
/execname == "soffice.bin"/
{
    self->on = vtimestamp;
}

sched:::off-cpu
/self->on/
{
    @time["<on cpu>"] = sum(vtimestamp - self->on);
    self->on = 0;
}

io:::wait-start
/execname == "soffice.bin"/
{
    self->wait = timestamp;
}

io:::wait-done
/self->wait/
{
    @io[args[2]->fi_name] = sum(timestamp - self->wait);
    @time["<I/O wait>"] = sum(timestamp - self->wait);
    self->wait = 0;
}

END
{
    printf("Time breakdown (milliseconds):\n");
    normalize(@time, 1000000);
    printa("  %-50s %15@d\n", @time);

    printf("\nI/O wait breakdown (milliseconds):\n");
```

```
    normalize(@io, 1000000);
    printa("  %-50s %15@d\n", @io);
}
```

Wenn Sie das Beispielskript während eines Kaltstarts von StarOffice ausführen, erhalten Sie folgende Ausgabe:

```
Time breakdown (milliseconds):
  <on cpu>                                                 3634
  <I/O wait>                                              13114

I/O wait breakdown (milliseconds):
  soffice.tmp                                                 0
  Office                                                      0
  unorc                                                       0
  sbasic.cfg                                                  0
  en                                                          0
  smath.cfg                                                   0
  toolboxlayout.xml                                           0
  sdraw.cfg                                                   0
  swriter.cfg                                                 0
  Linguistic.dat                                              0
  scalc.cfg                                                   0
  Views.dat                                                   0
  Store.dat                                                   0
  META-INF                                                    0
  Common.xml.tmp                                              0
  afm                                                         0
  libsimreg.so                                                1
  xiiimp.so.2                                                 3
  outline                                                     4
  Inet.dat                                                    6
  fontmetric                                                  6
  ...
  libucb1.so                                                 44
  libj641si_g.so                                             46
  libX11.so.4                                                46
  liblng641si.so                                             48
  swriter.db                                                 53
  libwrp641si.so                                             53
  liblocaledata_ascii.so                                     56
  libi18npool641si.so                                        65
  libdbtools2.so                                             69
  ofa64101.res                                               74
  libxcr641si.so                                             82
  libucpchelp1.so                                            83
  libsot641si.so                                             86
  libcppuhelper3C52.so                                       98
```

```
  libfwl641si.so                                          100
  libsb641si.so                                           104
  libcomphelp2.so                                         105
  libxo641si.so                                           106
  libucpfile1.so                                          110
  libcppu.so.3                                            111
  sw64101.res                                             114
  libdb-3.2.so                                            119
  libtk641si.so                                           126
  libdtransX11641si.so                                    127
  libgo641si.so                                           132
  libfwe641si.so                                          150
  libi18n641si.so                                         152
  libfwi641si.so                                          154
  libso641si.so                                           173
  libpsp641si.so                                          186
  libtl641si.so                                           189
  <unknown>                                               189
  libucbhelper1C52.so                                     195
  libutl641si.so                                          213
  libofa641si.so                                          216
  libfwk641si.so                                          229
  libsvl641si.so                                          261
  libcfgmgr2.so                                           368
  libsvt641si.so                                          373
  libvcl641si.so                                          741
  libsvx641si.so                                          885
  libsfx641si.so                                          993
  <none>                                                 1096
  libsw641si.so                                          1365
  applicat.rdb                                           1580
```

Wie diese Ausgabe zeigt, wird die meiste Wartezeit beim StarOffice-Kaltstart vom Warten auf E/A-Operationen verursacht (13,1 s Wartezeit auf E/A im Vergleich zu 3,6 s Wartezeit auf CPU-Operationen). Führen wir das Skript bei einem Warmstart von StarOffice aus, wird deutlich, dass die E/A-Zeit durch Seiten-Caching eliminiert wurde:

```
Time breakdown (milliseconds):
  <I/O wait>                                                0
  <on cpu>                                               2860

I/O wait breakdown (milliseconds):
  temp                                                      0
  soffice.tmp                                               0
  <unknown>                                                 0
  Office                                                    0
```

In der Kaltstart-Ausgabe sehen wir für die Datei `applicat.rdb` mehr E/A-Wartezeit als für jede andere Datei. Es ist anzunehmen, dass zahlreiche E/A-Zugriffe auf die Datei stattgefunden haben. Um diese weiter zu untersuchen, bedienen wir uns des folgenden D-Skripts:

```
io:::start
/execname == "soffice.bin" && args[2]->fi_name == "applicat.rdb"/
{
    @ = lquantize(args[2]->fi_offset != -1 ?
        args[2]->fi_offset / (1000 * 1024) : -1, 0, 1000);
}
```

In diesem Skript wird das Feld `fi_offset` der Struktur `fileinfo_t` verwendet, das uns Aufschluss über die Dateibereiche geben soll, auf die zugegriffen wird. Dabei gilt eine Granularität von einem Megabyte. Wenn Sie dieses Skript während eines Kaltstarts von StarOffice ausführen, erhalten Sie eine Ausgabe wie diese:

```
# dtrace -s ./applicat.d
dtrace: script ’./applicat.d’ matched 4 probes
^C


           value  ------------- Distribution ------------  count
             < 0 |                                        0
               0 |@@@                                     28
               1 |@@                                      17
               2 |@@@@                                    35
               3 |@@@@@@@@@@                              72
               4 |@@@@@@@@@@                              78
               5 |@@@@@@@@                                65
               6 |                                        0
```

Die Ausgabe sagt uns, dass nur auf die ersten sechs Megabyte der Datei zugegriffen wird, da die Datei möglicherweise sechs Megabyte groß ist. Außerdem deutet die Ausgabe darauf hin, dass nicht auf die gesamte Datei zugegriffen wird. Wollten wir nun die Kaltstartzeit für StarOffice verkürzen, wäre es hilfreich, das Zugriffsmuster der Datei zu verstehen. Wenn die benötigten Bereiche möglicherweise großteilig zusammenhängen, ließe sich die StarOffice-Kaltstartzeit beispielsweise mit einem Scout-Thread verbessern, der vor der Anwendung ausgeführt wird und die E/A-Zugriffe auf die Datei einleitet, noch bevor sie benötigt werden. (Dieser Ansatz ist dann besonders direkt, wenn der Zugriff auf die Datei über `mmap(2)` erfolgt.) Für die ca. 1,6 Sekunden Kaltstartzeit, die sich durch diese Strategie einsparen ließen, lohnt sich die zusätzliche Komplexität und Wartungslast in der Anwendung jedoch nicht. Auf jeden Fall ermöglichen die mit dem Provider `io` erhobenen Angaben ein genaues Verständnis der Vorteile, die durch einen solchen Aufwand zusätzlich erzielt werden könnten.

Stabilität

Der Provider `io` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Argumente	Evolving	Evolving	ISA

◆ ◆ ◆ KAPITEL 28

Der Provider `mib`

Der Provider `mib` stellt Prüfpunkte für Zähler in den Solaris-MIBs (Management Information Bases) zur Verfügung. MIB-Zähler werden in SNMP (Simple Network Management Protocol) zur Fernüberwachung heterogener Netzwerkgrößen verwendet. Die Zähler können auch mit den Befehlen `kstat(1M)` und `netstat(1M)` angezeigt werden. Der Provider `mib` hilft Ihnen, anormalem Netzwerkverhalten, das entweder durch lokale oder entfernte Netzwerküberwachung festgestellt wurde, schnell auf den Grund zu gehen.

Prüfpunkte

Der Provider `mib` stellt Prüfpunkte für Zähler aus verschiedenen MIBs zur Verfügung. In Tabelle 28–1 sind die Protokolle aufgeführt, die mithilfe des Providers `mib` instrumentierte MIBs exportieren. Die Tabelle enthält einen Verweis auf die eine MIB ganz oder teilweise beschreibende Dokumentation, den Namen der Kernelstatistik, über den auf die laufenden Zählungen (mit der Option `kstat(1M)` `-n` *Statistik*) zugegriffen werden kann, sowie einen Verweis auf die Tabelle, in der Sie die vollständige Definition der Prüfpunkte finden. Alle MIB-Zähler sind außerdem über die Option `-s` mit dem Befehl `netstat(1M)` abrufbar.

TABELLE 28–1 `mib`-Prüfpunkte

Protokoll	MIB-Beschreibung	Kernelstatistik	Tabelle der `mib`-Prüfpunkte
ICMP	RFC 1213	`icmp`	Tabelle 28–2
IP	RFC 1213	`ip`	Tabelle 28–3
IPsec	—	`ip`	Tabelle 28–4
IPv6	RFC 2465	—	Tabelle 28–5
SCTP	“SCTP MIB” (Internet-Entwurf)	`sctp`	Tabelle 28–7

TABELLE 28–1 `mib`-Prüfpunkte *(Fortsetzung)*

Protokoll	MIB-Beschreibung	Kernelstatistik	Tabelle der `mib`-Prüfpunkte
TCP	RFC 1213	`tcp`	Tabelle 28–8
UDP	RFC 1213	`udp`	Tabelle 28–9

TABELLE 28–2 `mib`-Prüfpunkte für ICMP

`icmpInAddrMaskReps`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Address Mask Reply“ ausgelöst wird.
`icmpInAddrMasks`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Address Mask Request“ ausgelöst wird.
`icmpInBadRedirects`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Redirect“ ausgelöst wird, in der ein Fehler festgestellt wurde (unbekannter ICMP-Code, Absender oder Ziel off-link usw.).
`icmpInCksumErrs`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung mit ungültiger Prüfsumme ausgelöst wird.
`icmpInDestUnreachs`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Destination Unreachable“ ausgelöst wird.
`icmpInEchoReps`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Echo Reply“ ausgelöst wird.
`icmpInEchos`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Echo Request“ ausgelöst wird.
`icmpInErrors`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung ausgelöst wird, in der ein ICMP-spezifischen Fehler (ungültige ICMP-Prüfsumme, ungültige Länge usw.) festgestellt wurde.
`icmpInFragNeeded`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Destination Unreachable (Fragmentation Needed)“ ausgelöst wird, die darauf hindeutet, dass ein gesendetes Paket verloren gegangen ist, da es eine MTU-Größe überschritten hat und das Flag „Don't Fragment“ gesetzt war.
`icmpInMsgs`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung ausgelöst wird. Immer, wenn dieser Prüfpunkt ausgelöst wird, kann auch der Prüfpunkt `icmpInErrors` ausgelöst werden, wenn ein ICMP-spezifischer Fehler in der Meldung festgestellt wurde.
`icmpInOverflows`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung ausgelöst wird, wenn die Meldung anschließend aufgrund von Pufferplatzmangel verworfen wird.
`icmpInParmProbs`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Parameter Problem“ ausgelöst wird.

TABELLE 28–2 `mib`-Prüfpunkte für ICMP *(Fortsetzung)*

`icmpInRedirects`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Redirect“ ausgelöst wird.
`icmpInSrcQuenchs`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Source Quench“ ausgelöst wird.
`icmpInTimeExcds`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Time Exceeded“ ausgelöst wird.
`icmpInTimestampReps`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Timestamp Reply“ ausgelöst wird.
`icmpInTimestamps`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung des Typs „Timestamp Request“ ausgelöst wird.
`icmpInUnknowns`	Prüfpunkt, der mit jedem Empfang einer ICMP-Meldung unbekannten Typs ausgelöst wird.
`icmpOutAddrMaskReps`	Prüfpunkt, der mit jeder Aussendung einer ICMP-Meldung des Typs „Address Mask Reply“ ausgelöst wird.
`icmpOutDestUnreachs`	Prüfpunkt, der mit jeder Aussendung einer ICMP-Meldung des Typs „Destination Unreachable“ ausgelöst wird.
`icmpOutDrops`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine ausgehende ICMP-Meldung aus irgendeinem Grund (Speicherzuordnungsfehler, Broadcast/Multicast-Absender oder -Ziel u. Ä.) verworfen wird.
`icmpOutEchoReps`	Prüfpunkt, der mit jeder Aussendung einer ICMP-Meldung des Typs „Echo Reply“ ausgelöst wird.
`icmpOutErrors`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine ICMP-Meldung aufgrund von innerhalb von ICMP festgestellten Problemen (z. B. Pufferplatzmangel) nicht abgesendet wird. Dieser Prüfpunkt wird nicht ausgelöst, wenn außerhalb der ICMP-Schicht Fehler festgestellt werden (z. B. dass IP das resultierende Datagramm nicht routen kann).
`icmpOutFragNeeded`	Prüfpunkt, der mit jeder Aussendung einer ICMP-Meldung des Typs „Destination Unreachable (Fragmentation Needed)“ ausgelöst wird.
`icmpOutMsgs`	Prüfpunkt, der mit jeder Aussendung einer ICMP-Meldung ausgelöst wird. Immer, wenn dieser Prüfpunkt ausgelöst wird, kann auch der Prüfpunkt `icmpOutErrors` ausgelöst werden, wenn festgestellt wurde, dass die Meldung ICMP-spezifische Fehler enthält.
`icmpOutParmProbs`	Prüfpunkt, der mit jeder Aussendung einer ICMP-Meldung des Typs „Parameter Problem“ ausgelöst wird.
`icmpOutRedirects`	Prüfpunkt, der mit jeder Aussendung einer ICMP-Meldung des Typs „Redirect“ ausgelöst wird. Für Hosts wird dieser Prüfpunkt niemals ausgelöst, da Hosts keine Weiterleitungen (Redirect) aussenden.

TABELLE 28–2 `mib`-Prüfpunkte für ICMP *(Fortsetzung)*

`icmpOutTimeExcds`	Prüfpunkt, der mit jeder Aussendung einer ICMP-Meldung des Typs „Time Exceeded“ ausgelöst wird.
`icmpOutTimestampReps`	Prüfpunkt, der mit jeder Aussendung einer ICMP-Meldung des Typs „Timestamp Reply“ ausgelöst wird.

TABELLE 28–3 `mib`-Prüfpunkte für IP

`ipForwDatagrams`	Prüfpunkt, der mit jedem Empfang eines Datagramms ausgelöst wird, dessen IP-Endziel nicht dieses System ist, und versucht wird, eine Route zur Weiterleitung des Datagramms an jenes Endziel zu finden. Auf Systemen, die nicht als IP-Gateway fungieren, wird dieser Prüfpunkt nur für Pakete ausgelöst, die per Source-Routing durch diese Maschine geleitet werden und bei denen die Verarbeitung der Source-Route-Option erfolgreich verlief.
`ipForwProhibits`	Prüfpunkt, der mit jedem Empfang eines Datagramms ausgelöst wird, dessen IP-Endziel nicht dieses System ist, aber, da das System nicht als Router zugelassen ist, nicht versucht wird, eine Route zur Weiterleitung des Datagramms an jenes Endziel zu finden.
`ipFragCreates`	Prüfpunkt, der immer dann ausgelöst wird, wenn bei einer Fragmentierung ein IP-Datagrammfragment erzeugt wird.
`ipFragFails`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IP-Datagramm verworfen wird, da es nicht fragmentiert werden konnte (z. B. weil die Fragmentierung erforderlich, aber das Flag „Don't Fragment“ gesetzt war).
`ipFragOKs`	Prüfpunkt, der mit jeder erfolgreichen Fragmentierung eines IP-Datagramms ausgelöst wird.
`ipInCksumErrs`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Eingangsdatagramm aufgrund einer ungültigen IP-Header-Prüfsumme verworfen wird.
`ipInDelivers`	Prüfpunkt, der mit jeder erfolgreichen Lieferung eines Eingangsdatagramms an IP-Benutzerprotokolle, einschließlich ICMP, ausgelöst wird.
`ipInDiscards`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IP-Eingangsdatagramm aus Gründen verworfen wird, die nicht mit dem Paket in Zusammenhang stehen (z. B. Pufferplatzmangel). Dieser Prüfpunkt wird nicht für Datagramme ausgelöst, die verworfen werden, während Sie auf die Defragmentierung warten.
`ipInHdrErrors`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Eingangsdatagramm aufgrund eines Fehlers im IP-Header verworfen wird. Dabei kann es sich u. a. um Versionskonflikte, Formatfehler, Lebenszeitüberschreitung oder Fehler handeln, die bei der Verarbeitung von IP-Optionen festgestellt werden.
`ipInIPv6`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IPv6-Paket fälschlicherweise in einer IPv4-Warteschlange ankommt.

TABELLE 28–3 `mib`-Prüfpunkte für IP *(Fortsetzung)*

`ipInReceives`	Prüfpunkt, der bei jedem - selbst fälschlicherweise erfolgtem - Empfang eines Datagramms von einer Schnittstelle ausgelöst wird.
`ipInUnknownProtos`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein lokal adressiertes Datagramm erfolgreich empfangen, aber anschließend aufgrund eines unbekannten oder nicht unterstützten Protokolls verworfen wird.
`ipOutDiscards`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IP-Ausgangsdatagramm aus Gründen verworfen wird, die nicht mit dem Paket in Zusammenhang stehen (z. B. Pufferplatzmangel). Dieser Prüfpunkt wird für jedes im MIB-Zähler `ipForwDatagrams` berücksichtigte Datenpaket ausgelöst, sofern das Paket solch ein (wahlfreies) Verwurfkriterium erfüllt.
`ipOutIPv6`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IPv6-Paket über eine IPv4-Verbindung gesendet wird.
`ipOutNoRoutes`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IP-Datagramm verworfen wird, da keine Route für die Übertragung an das Ziel gefunden werden konnte. Dieser Prüfpunkt wird für jedes im MIB-Zähler `ipForwDatagrams` berücksichtigte Datenpaket ausgelöst, sofern das Paket dieses „no-route"-Kriterium erfüllt. Dieser Prüfpunkt wird auch für Datagramme ausgelöst, die nicht geroutet werden können, da alle Standard-Gateways ausgefallen sind.
`ipOutRequests`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IP-Datagramm zur Übertragung von lokalen IP-Benutzerprotokollen (einschl. ICMP) an IP übergeben wird. Beachten Sie, dass dieser Prüfpunkt nicht für die vom MIB-Zähler `ipForwDatagrams` berücksichtigten Datenpakete ausgelöst wird.
`ipOutSwitchIPv6`	Prüfpunkt, der bei jedem Wechsel von IPv4 zu IPv6 als IP-Protokoll für eine Verbindung ausgelöst wird.
`ipReasmDuplicates`	Prüfpunkt, der immer dann ausgelöst wird, wenn der IP-Defragmentierungsalgorithmus bestimmt, dass ein IP-Fragment *nur* zuvor empfangene Daten enthält.
`ipReasmFails`	Prüfpunkt, der immer dann ausgelöst wird, wenn der IP-Defragmentierungsalgorithmus einen Fehler feststellt. Dieser Prüfpunkt wird nicht unbedingt für jedes verworfene IP-Fragment ausgelöst, da einige Algorithmen (insbesondere der Algorithmus in RFC 815) Fragmente aus den Augen verlieren können, indem sie diese direkt beim Empfang kombinieren.
`ipReasmOKs`	Prüfpunkt, der mit jeder erfolgreichen Defragmentierung eines IP-Datagramms ausgelöst wird.
`ipReasmPartDups`	Prüfpunkt, der immer dann ausgelöst wird, wenn der IP-Defragmentierungsalgorithmus bestimmt, dass ein IP-Fragment sowohl zuvor empfangene als auch neue Daten enthält.
`ipReasmReqds`	Prüfpunkt, der mit jedem Empfang eines IP-Fragments ausgelöst wird, das wieder zusammengesetzt werden muss.

TABELLE 28–4 `mib`-Prüfpunkte für IPsec

`ipsecInFailed`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein empfangenes Paket wegen Nichterfüllung der angegebenen IPsec-Richtlinie verworfen wird.
`ipsecInSucceeded`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein empfangenes Paket die angegebene IPsec-Richtlinie erfüllt und die Verarbeitung fortgesetzt werden darf.

TABELLE 28–5 `mib`-Prüfpunkte für IPv6

`ipv6ForwProhibits`	Prüfpunkt, der mit jedem Empfang eines IPv6-Datagramms ausgelöst wird, dessen IPv6-Endziel nicht dieses System ist, aber, da das System nicht als Router zugelassen ist, nicht versucht wird, eine Route zur Weiterleitung des Datagramms an jenes Endziel zu finden.
`ipv6IfIcmpBadHoplimit`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-NDP-Meldung (NDP, Neighbor Discovery Protocol) ausgelöst wird, für das ein Hop-Limit unter dem festgelegten Maximum festgestellt wird. Diese Meldungen stammen möglicherweise nicht von einem Nachbarn (am selben Netzwerk hängender Knoten) und werden deshalb verworfen.
`ipv6IfIcmpInAdminProhibs`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Destination Unreachable (Communication Administratively Prohibited)“ ausgelöst wird.
`ipv6IfIcmpInBadNeighborAdvertisements`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Neighbor Advertisement“ ausgelöst wird, die auf irgendeine Weise Fehler enthält.
`ipv6IfIcmpInBadNeighborSolicitations`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Neighbor Solicit“ ausgelöst wird, die auf irgendeine Weise Fehler enthält.
`ipv6IfIcmpInBadRedirects`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Redirect“ ausgelöst wird, die auf irgendeine Weise Fehler enthält.
`ipv6IfIcmpInDestUnreachs`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Destination Unreachable“ ausgelöst wird.
`ipv6IfIcmpInEchoReplies`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Echo Reply“ ausgelöst wird.

TABELLE 28–5 `mib`-Prüfpunkte für IPv6 *(Fortsetzung)*

`ipv6IfIcmpInEchos`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Echo Request“ ausgelöst wird.
`ipv6IfIcmpInErrors`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung ausgelöst wird, in der ein ICMPv6-spezifischer Fehler (ungültige ICMPv6-Prüfsumme, ungültige Länge usw.) festgestellt wurde.
`ipv6IfIcmpInGroupMembBadQueries`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Group Membership Query“ ausgelöst wird, die auf irgendeine Weise Fehler enthält.
`ipv6IfIcmpInGroupMembBadReports`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Group Membership Report“ ausgelöst wird, die auf irgendeine Weise Fehler enthält.
`ipv6IfIcmpInGroupMembOurReports`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Group Membership Report“ ausgelöst wird.
`ipv6IfIcmpInGroupMembQueries`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Group Membership Query“ ausgelöst wird.
`ipv6IfIcmpInGroupMembReductions`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Group Membership Reduction“ ausgelöst wird.
`ipv6IfIcmpInGroupMembResponses`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Group Membership Response“ ausgelöst wird.
`ipv6IfIcmpInGroupMembTotal`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-MLD-Meldung (MLD, Multicast Listener Discovery) ausgelöst wird.
`ipv6IfIcmpInMsgs`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung ausgelöst wird. Wenn dieser Prüfpunkt ausgelöst wird, kann auch der Prüfpunkt `ipv6IfIcmpInErrors` ausgelöst werden, wenn die Meldung einen ICMPv6-spezifischen Fehler enthält.
`ipv6IfIcmpInNeighborAdvertisements`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Neighbor Advertisement“ ausgelöst wird.

TABELLE 28–5 `mib`-Prüfpunkte für IPv6 *(Fortsetzung)*

`ipv6IfIcmpInNeighborSolicits`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Neighbor Solicit“ ausgelöst wird.
`ipv6IfIcmpInOverflows`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung ausgelöst wird, wenn die Meldung anschließend aufgrund von Pufferplatzmangel verworfen wird.
`ipv6IfIcmpInParmProblems`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Parameter Problem“ ausgelöst wird.
`ipv6IfIcmpInRedirects`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Redirect“ ausgelöst wird.
`ipv6IfIcmpInRouterAdvertisements`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Router Advertisement“ ausgelöst wird.
`ipv6IfIcmpInRouterSolicits`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Router Solicit“ ausgelöst wird.
`ipv6IfIcmpInTimeExcds`	Prüfpunkt, der mit jedem Empfang einer ICMPv6-Meldung des Typs „Time Exceeded“ ausgelöst wird.
`ipv6IfIcmpOutAdminProhibs`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Destination Unreachable (Communication Administratively Prohibited)“ ausgelöst wird.
`ipv6IfIcmpOutDestUnreachs`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Destination Unreachable“ ausgelöst wird.
`ipv6IfIcmpOutEchoReplies`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Echo Reply“ ausgelöst wird.
`ipv6IfIcmpOutEchos`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Echo“ ausgelöst wird.
`ipv6IfIcmpOutErrors`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine ICMPv6-Meldung aufgrund von innerhalb von ICMPv6 festgestellten Problemen (z. B. Pufferplatzmangel) nicht abgesendet wird. Dieser Prüfpunkt wird nicht ausgelöst, wenn außerhalb der ICMPv6-Schicht Fehler festgestellt werden (z. B. dass IPv6 das resultierende Datagramm nicht routen kann).

TABELLE 28–5 `mib`-Prüfpunkte für IPv6 *(Fortsetzung)*

`ipv6IfIcmpOutGroupMembQueries`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Group Membership Query“ ausgelöst wird.
`ipv6IfIcmpOutGroupMembReductions`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Group Membership Reduction“ ausgelöst wird.
`ipv6IfIcmpOutGroupMembResponses`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Group Membership Response“ ausgelöst wird.
`ipv6IfIcmpOutMsgs`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung ausgelöst wird. Wenn dieser Prüfpunkt ausgelöst wird, kann auch der Prüfpunkt `ipv6IfIcmpOutErrors` ausgelöst werden, wenn die Meldung ICMPv6-spezifische Fehler enthält.
`ipv6IfIcmpOutNeighborAdvertisements`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Neighbor Advertisement“ ausgelöst wird.
`ipv6IfIcmpOutNeighborSolicits`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Neighbor Solicitation“ ausgelöst wird.
`ipv6IfIcmpOutParmProblems`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Parameter Problem“ ausgelöst wird.
`ipv6IfIcmpOutPktTooBigs`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Packet Too Big“ ausgelöst wird.
`ipv6IfIcmpOutRedirects`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Redirect“ ausgelöst wird. Für Hosts wird dieser Prüfpunkt niemals ausgelöst, da Hosts keine Weiterleitungen (Redirect) aussenden.
`ipv6IfIcmpOutRouterAdvertisements`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Router Advertisement“ ausgelöst wird.
`ipv6IfIcmpOutRouterSolicits`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Router Solicit“ ausgelöst wird.
`ipv6IfIcmpOutTimeExcds`	Prüfpunkt, der mit jeder Aussendung einer ICMPv6-Meldung des Typs „Time Exceeded“ ausgelöst wird.

TABELLE 28–5 `mib`-Prüfpunkte für IPv6 *(Fortsetzung)*

`ipv6InAddrErrors`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Eingangsdatagramm verworfen wird, weil die IPv6-Adresse im Zielfeld des IPv6-Headers keine gültige Adresse für diesen Empfänger ist. Dieser Prüfpunkt wird bei ungültigen (z. B. : : 0) und nicht unterstützten Adressen ausgelöst (z. B. Adressen mit nicht zugewiesenen Präfixen). Bei Systemen, die nicht als IPv6-Router konfiguriert sind und folglich keine Datagramme weiterleiten, wird dieser Prüfpunkt für Datagramme ausgelöst, die verworfen werden, da die Zieladresse keine lokale Adresse ist.
`ipv6InDelivers`	Prüfpunkt, der mit jeder erfolgreichen Lieferung eines Eingangsdatagramms an IPv6-Benutzerprotokolle (einschließlich ICMPv6) ausgelöst wird.
`ipv6InDiscards`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IPv6-Eingangsdatagramm aus Gründen verworfen wird, die nicht mit dem Paket in Zusammenhang stehen (z. B. Pufferplatzmangel). Dieser Prüfpunkt wird nicht für Datagramme ausgelöst, die verworfen werden, während Sie auf die Defragmentierung warten.
`ipv6InHdrErrors`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Eingangsdatagramm aufgrund eines Fehlers im IPv6-Header verworfen wird. Dabei kann es sich u. a. um Versionskonflikte, Formatfehler, Überschreitung der Hop-Anzahl oder Fehler handeln, die bei der Verarbeitung von IPv6-Optionen festgestellt werden.
`ipv6InIPv4`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IPv4-Paket fälschlicherweise in einer IPv6-Warteschlange ankommt.
`ipv6InMcastPkts`	Prüfpunkt, der bei jedem Empfang eines IPv6-Multicast-Pakets ausgelöst wird.
`ipv6InNoRoutes`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein geroutetes IPv6-Datagramm verworfen wird, da keine Route für die Übertragung an das Ziel gefunden werden konnte. Dieser Prüfpunkt wird *nur* für Datenpakete ausgelöst, die von einer externen Quelle stammen.
`ipv6InReceives`	Prüfpunkt, der bei jedem - selbst fälschlicherweise erfolgtem - Empfang eines IPv6-Datagramms von einer Schnittstelle ausgelöst wird.

TABELLE 28–5 `mib`-Prüfpunkte für IPv6 *(Fortsetzung)*

`ipv6InTooBigErrors`	Prüfpunkt, der mit jedem Empfang eines Fragments ausgelöst wird, das die maximale Fragmentgröße übersteigt.
`ipv6InTruncatedPkts`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Eingangsdatagramm verworfen wird, da der Datagramm-Frame nicht genügend Daten enthielt.
`ipv6InUnknownProtos`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein lokal adressiertes IPv6-Datagramm erfolgreich empfangen, aber anschließend aufgrund eines unbekannten oder nicht unterstützten Protokolls verworfen wird.
`ipv6OutDiscards`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IPv6-Ausgangsdatagramm aus Gründen verworfen wird, die nicht mit dem Paket in Zusammenhang stehen (z. B. Pufferplatzmangel). Dieser Prüfpunkt wird für jedes im MIB-Zähler `ipv6OutForwDatagrams` berücksichtigte Datenpaket ausgelöst, sofern das Paket solch ein (wahlfreies) Verwurfkriterium erfüllt.
`ipv6OutForwDatagrams`	Prüfpunkt, der mit jedem Empfang eines Datagramms ausgelöst wird, dessen IPv6-Endziel nicht dieses System ist, und versucht wird, eine Route zur Weiterleitung des Datagramms an jenes Endziel zu finden. Auf einem System, das nicht als IPv6-Gateway fungiert, wird dieser Prüfpunkt nur für Pakete ausgelöst, die per Source-Routing durch dieses System geleitet werden und bei denen die Verarbeitung der Source-Route-Option erfolgreich verlief.
`ipv6OutFragCreates`	Prüfpunkt, der immer dann ausgelöst wird, wenn bei einer Fragmentierung ein IPv6-Datagrammfragment erzeugt wird.
`ipv6OutFragFails`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IPv6-Datagramm verworfen wird, da es nicht fragmentiert werden konnte, da beispielsweise das Flag „Don't Fragment“ gesetzt war.
`ipv6OutFragOKs`	Prüfpunkt, der mit jeder erfolgreichen Fragmentierung eines IPv6-Datagramms ausgelöst wird.
`ipv6OutIPv4`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IPv6-Paket über eine IPv4-Verbindung gesendet wird.
`ipv6OutMcastPkts`	Prüfpunkt, der bei jeder Aussendung eines Multicast-Pakets ausgelöst wird.

TABELLE 28–5 `mib`-Prüfpunkte für IPv6 *(Fortsetzung)*

`ipv6OutNoRoutes`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IPv6-Datagramm verworfen wird, da keine Route für die Übertragung an das Ziel gefunden werden konnte. Dieser Prüfpunkt wird *nicht* für Datenpakete ausgelöst, die von einer externen Quelle stammen.
`ipv6OutRequests`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein IPv6-Datagramm zur Übertragung von lokalen IPv6-Benutzerprotokollen (einschl. ICMPv6) an IPv6 übergeben wird. Dieser Prüfpunkt wird für die im MIB-Zähler `ipv6ForwDatagrams` berücksichtigten Pakete nicht ausgelöst.
`ipv6OutSwitchIPv4`	Prüfpunkt, der bei jedem Wechsel von IPv6 zu IPv4 als IP-Protokoll für eine Verbindung ausgelöst wird.
`ipv6ReasmDuplicates`	Prüfpunkt, der immer dann ausgelöst wird, wenn der IPv6-Defragmentierungsalgorithmus bestimmt, dass ein IPv6-Fragment *nur* zuvor empfangene Daten enthält.
`ipv6ReasmFails`	Prüfpunkt, der immer dann ausgelöst wird, wenn der IPv6-Defragmentierungsalgorithmus einen Fehler feststellt. Dieser Prüfpunkt wird nicht unbedingt für jedes verworfene IPv6-Fragment ausgelöst, da einige Algorithmen Fragmente aus den Augen verlieren können, indem sie diese direkt beim Empfang kombinieren.
`ipv6ReasmOKs`	Prüfpunkt, der mit jeder erfolgreichen Defragmentierung eines IPv6-Datagramms ausgelöst wird.
`ipv6ReasmPartDups`	Prüfpunkt, der immer dann ausgelöst wird, wenn der IPv6-Defragmentierungsalgorithmus bestimmt, dass ein IPv6-Fragment sowohl zuvor empfangene als auch neue Daten enthält.
`ipv6ReasmReqds`	Prüfpunkt, der mit jedem Empfang eines IPv6-Fragments ausgelöst wird, das wieder zusammengesetzt werden muss.

TABELLE 28–6 `mib`-Prüfpunkte für Raw IP

`rawipInCksumErrs`	Prüfpunkt, der bei jedem Empfang eines Raw-IP-Pakets mit einer ungültigen IP-Prüfsumme ausgelöst wird.
`rawipInDatagrams`	Prüfpunkt, der bei jedem Empfang eines Raw-IPv6-Pakets ausgelöst wird.

TABELLE 28–6 `mib`-Prüfpunkte für Raw IP *(Fortsetzung)*

`rawipInErrors`	Prüfpunkt, der bei jedem Empfang eines Raw-IP-Pakets ausgelöst wird, das auf irgendeine Weise Fehler enthält.
`rawipInOverflows`	Prüfpunkt, der mit jedem Empfang eines Raw-IP-Pakets ausgelöst wird, wenn das Paket anschließend aufgrund von Pufferplatzmangel verworfen wird.
`rawipOutDatagrams`	Prüfpunkt, der bei jeder Aussendung eines Raw-IPv6-Pakets ausgelöst wird.
`rawipOutErrors`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Raw-IP-Paket aufgrund einer Fehlerbedingung (in der Regel wegen eines Formatfehlers des IP-Pakets) nicht gesendet wurde.

TABELLE 28–7 `mib`-Prüfpunkte für SCTP

`sctpAborted`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine SCTP-Association von einem beliebigen Status mittels des Grundelements ABORT direkt in den Status CLOSED übergegangen ist. Dies deutet auf eine sofortige Schließung der Verbindung hin.
`sctpActiveEstab`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine SCTP-Association aus dem Status COOKIE-ECHOED direkt in den Status ESTABLISHED übergegangen ist. Dies deutet darauf hin, dass die obere Ebene den Association-Versuch eingeleitet hat.
`sctpChecksumError`	Prüfpunkt, der bei jedem Empfang eines SCTP-Pakets mit ungültiger Prüfsumme von einem Peer ausgelöst wird.
`sctpCurrEstab`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine SCTP-Association beim Lesen des MIB-Zählers `sctpCurrEstab` gezählt wird. Gezählt werden SCTP-Associations, deren aktueller Status ESTABLISHED, SHUTDOWN-RECEIVED oder SHUTDOWN-PENDING ist.
`sctpFragUsrMsgs`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine Benutzernachricht aufgrund der MTU fragmentiert werden muss.
`sctpInClosed`	Prüfpunkt, der bei jedem Empfang von Daten an einer geschlossenen SCTP-Association ausgelöst wird.
`sctpInCtrlChunks`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpInCtrlChunks` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.
`sctpInDupAck`	Prüfpunkt, der bei jedem Empfang einer doppelten ACK (Bestätigung) ausgelöst wird.
`sctpInInvalidCookie`	Prüfpunkt, der mit jedem Empfang eines ungültigen Cookies ausgelöst wird.

TABELLE 28–7 `mib`-Prüfpunkte für SCTP *(Fortsetzung)*

`sctpInOrderChunks`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpInOrderChunks` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.
`sctpInSCTPPkts`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpInSCTPPkts` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.
`sctpInUnorderChunks`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpInUnorderChunks` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.
`sctpListenDrop`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine eingehende Verbindung aus irgendeinem Grund verworfen wird.
`sctpOutAck`	Prüfpunkt, der bei jeder Aussendung einer SACK (selektiven Bestätigung) ausgelöst wird.
`sctpOutAckDelayed`	Prüfpunkt, der immer dann ausgelöst wird, wenn für eine SCTP-Association eine verzögerte Bestätigungsverarbeitung stattfindet. Jede im Rahmen der verzögerten Bestätigungsverarbeitung gesendete ACK löst den Prüfpunkt `sctpOutAck` aus.
`sctpOutCtrlChunks`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpOutCtrlChunks` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.
`sctpOutOfBlue`	Prüfpunkt, der bei jedem Empfang eines an sich fehlerfreien SCTP-Pakets ausgelöst wird, für das der Empfänger die zugehörige Association jedoch nicht ermitteln kann.
`sctpOutOrderChunks`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpOutOrderChunks` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.
`sctpOutSCTPPkts`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpOutSCTPPkts` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.

TABELLE 28–7 `mib`-Prüfpunkte für SCTP *(Fortsetzung)*

`sctpOutUnorderChunks`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpOutUnorderChunks` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.
`sctpOutWinProbe`	Prüfpunkt, der bei jeder Aussendung einer Window Probe (Prüfung auf Empfangsbereitschaft) ausgelöst wird.
`sctpOutWinUpdate`	Prüfpunkt, der bei jeder Aussendung einer Window-Aktualisierung ausgelöst wird.
`sctpPassiveEstab`	Prüfpunkt, der bei jedem direkten Übergang von SCTP-Associations aus dem Status ESTABLISHED in den Status CLOSED ausgelöst wird. Der entfernte Endpunkt hat den Association-Versuch eingeleitet.
`sctpReasmUsrMsgs`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpReasmUsrMsg` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.
`sctpRetransChunks`	Prüfpunkt, der immer dann ausgelöst wird, wenn der MIB-Zähler `sctpRetransChunks` entweder wegen einer expliziten Abfrage des MIB-Zählers oder aufgrund der Schließung einer SCTP-Verbindung aktualisiert wird. Der Wert, um den der MIB-Zähler zu erhöhen ist, befindet sich in `args[0]`.
`sctpShutdowns`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine SCTP-Association entweder aus dem Status SHUTDOWN-SENT oder SHUTDOWN-ACK-SENT direkt in den Status CLOSED übergeht. Dies deutet auf eine sofortige Schließung der Association hin.
`sctpTimHeartBeatDrop`	Prüfpunkt, der mit jedem Abbruch einer SCTP-Association aufgrund eines fehlgeschlagenen Empfangs einer Heartbeat-Bestätigung ausgelöst wird.
`sctpTimHeartBeatProbe`	Prüfpunkt, der mit jeder Aussendung eines SCTP-Heartbeat ausgelöst wird.
`sctpTimRetrans`	Prüfpunkt, der bei jeder timer-basierten Retransmit-Verarbeitung für eine Association ausgelöst wird.
`sctpTimRetransDrop`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine Association abgebrochen wird, da eine timer-basierte Retransmit-Verarbeitung über einen längeren Zeitraum nicht möglich war.

TABELLE 28–8 `mib`-Prüfpunkte für TCP

`tcpActiveOpens`	Prüfpunkt, der bei jedem direkten Übergang einer TCP-Verbindung aus dem Status CLOSED in den Status SYN-SENT ausgelöst wird.

TABELLE 28–8 `mib`-Prüfpunkte für TCP *(Fortsetzung)*

`tcpAttemptFails`	Prüfpunkt, der bei jedem direkten Übergang einer TCP-Verbindung aus dem Status SYN_SENT oder SYN-RCVD in den Status CLOSED oder aus dem Status SYN_RCVD in den Status LISTEN ausgelöst wird.
`tcpCurrEstab`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine TCP-Verbindung beim Lesen des MIB-Zählers `tcpCurrEstab` gezählt wird. Gezählt werden TCP-Verbindungen mit dem aktuellen Status ESTABLISHED oder CLOSE_WAIT.
`tcpEstabResets`	Prüfpunkt, der bei jedem direkten Übergang einer TCP-Verbindung aus dem Status ESTABLISHED oder CLOSE_WAIT in den Status CLOSED ausgelöst wird.
`tcpHalfOpenDrop`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine Verbindung aufgrund einer vollen Warteschlange mit Verbindungen im Status SYN_RCVD verworfen wird.
`tcpInAckBytes`	Prüfpunkt, der mit jedem Empfang einer ACK für zuvor gesendete Daten ausgelöst wird. Die Anzahl der bestätigten Byte wird mit `args[0]` übergeben.
`tcpInAckSegs`	Prüfpunkt, der mit jedem Empfang einer ACK für ein zuvor gesendetes Segment ausgelöst wird.
`tcpInAckUnsent`	Prüfpunkt, der mit jedem Empfang einer ACK für ein nicht gesendetes Segment ausgelöst wird.
`tcpInClosed`	Prüfpunkt, der nach jedem Empfang von Daten für eine Verbindung in einem schließenden Status ausgelöst wird.
`tcpInDataDupBytes`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Segment empfangen wird und alle Daten in dem Segment zuvor bereits empfangen wurden. Die Anzahl der Byte in dem doppelt vorliegenden Segment wird mit `args[0]` übergeben.
`tcpInDataDupSegs`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Segment empfangen wird und alle Daten in dem Segment zuvor bereits empfangen wurden. Die Anzahl der Byte in dem doppelt vorliegenden Segment wird mit `args[0]` übergeben.
`tcpInDataInorderBytes`	Prüfpunkt, der ausgelöst wird, wenn Daten empfangen werden und *alle* Daten vor der Sequenznummer der neuen Daten bereits zuvor empfangen wurden. Die Anzahl der reihenfolgengetreu empfangenen Byte wird mit `args[0]` übergeben.
`tcpInDataInorderSegs`	Prüfpunkt, der ausgelöst wird, wenn ein Segment empfangen wird und *alle* Daten vor der Sequenznummer des neuen Segments bereits zuvor empfangen wurden.
`tcpInDataPartDupBytes`	Prüfpunkt, der ausgelöst wird, wenn ein Segment empfangen wird und einige der Daten im Segment zuvor bereits empfangen wurden, andere Daten im Segment aber neu sind. Die Anzahl der doppelt vorliegenden Byte wird mit `args[0]` übergeben.

TABELLE 28–8 `mib`-Prüfpunkte für TCP *(Fortsetzung)*

`tcpInDataPartDupSegs`	Prüfpunkt, der ausgelöst wird, wenn ein Segment empfangen wird und einige der Daten im Segment zuvor bereits empfangen wurden, andere Daten im Segment aber neu sind. Die Anzahl der doppelt vorliegenden Byte wird mit `args[0]` übergeben.
`tcpInDataPastWinBytes`	Prüfpunkt, der mit jedem Empfang von Daten ausgelöst wird, die außerhalb des aktuellen Empfangsfensters liegen. Die Anzahl der Byte ist in `args[0]` enthalten.
`tcpInDataPastWinSegs`	Prüfpunkt, der mit jedem Empfang eines Segments ausgelöst wird, das außerhalb des aktuellen Empfangsfensters liegt.
`tcpInDataUnorderBytes`	Prüfpunkt, der ausgelöst wird, wenn Daten empfangen werden und einige der Daten vor der Sequenznummer der neuen Daten fehlen. Die Anzahl der nicht reihenfolgengetreu empfangenen Byte wird mit `args[0]` übergeben.
`tcpInDataUnorderSegs`	Prüfpunkt, der ausgelöst wird, wenn ein Segment empfangen wird und einige der Daten vor der Sequenznummer der neuen Daten fehlen.
`tcpInDupAck`	Prüfpunkt, der bei jedem Empfang einer doppelten ACK (Bestätigung) ausgelöst wird.
`tcpInErrs`	Prüfpunkt, der immer dann ausgelöst wird, wenn in einem empfangenen Segment ein TCP-Fehler (z. B. eine ungültige TCP-Prüfsumme) gefunden wird.
`tcpInSegs`	Prüfpunkt, der mit jedem Empfang eines Segments ausgelöst wird, selbst wenn in diesem Segment später ein Fehler festgestellt wird, der die weitere Verarbeitung verhindert.
`tcpInWinProbe`	Prüfpunkt, der bei jedem Empfang einer Window Probe (Prüfung auf Empfangsbereitschaft) ausgelöst wird.
`tcpInWinUpdate`	Prüfpunkt, der bei jedem Empfang einer Window-Aktualisierung ausgelöst wird.
`tcpListenDrop`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine eingehende Verbindung aufgrund einer vollen Listen-Warteschlange verworfen wird.
`tcpListenDropQ0`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine Verbindung aufgrund einer vollen Warteschlange mit Verbindungen im Status SYN_RCVD verworfen wird.
`tcpOutAck`	Prüfpunkt, der mit jeder Aussendung einer ACK ausgelöst wird.
`tcpOutAckDelayed`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine ACK nach einer Verzögerung gesendet wird.
`tcpOutControl`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein SYN, FIN oder RST gesendet wird.
`tcpOutDataBytes`	Prüfpunkt, der mit jeder Datensendung ausgelöst wird. Die Anzahl der gesendeten Byte ist in `args[0]` enthalten.

TABELLE 28–8 `mib`-Prüfpunkte für TCP *(Fortsetzung)*

`tcpOutDataSegs`	Prüfpunkt, der mit jeder Sendung eines Segments ausgelöst wird.
`tcpOutFastRetrans`	Prüfpunkt, der mit jeder erneuten Übertragung eines Segments im Rahmen des „Fast-Retransmit“-Algorithmus ausgelöst wird.
`tcpOutRsts`	Prüfpunkt, der mit jeder Sendung eines Segments mit gesetztem RST-Flag ausgelöst wird.
`tcpOutSackRetransSegs`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Segment über eine Verbindung mit aktivierter SACK (selektive Bestätigung) erneut übertragen wird.
`tcpOutSegs`	Prüfpunkt, der mit jeder Sendung eines Segments ausgelöst wird, das mindestens ein nicht neu übertragenes Byte enthält.
`tcpOutUrg`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Segment mit gesetztem URG-Flag und gültigem Dringlichkeitszeiger gesendet wird.
`tcpOutWinProbe`	Prüfpunkt, der bei jeder Aussendung einer Window Probe (Prüfung auf Empfangsbereitschaft) ausgelöst wird.
`tcpOutWinUpdate`	Prüfpunkt, der bei jeder Aussendung einer Window-Aktualisierung ausgelöst wird.
`tcpPassiveOpens`	Prüfpunkt, der bei jedem direkten Übergang einer TCP-Verbindung aus dem Status LISTEN in den Status SYN_RCVD ausgelöst wird.
`tcpRetransBytes`	Prüfpunkt, der mit jeder erneuten Übertragung von Daten ausgelöst wird. Die Anzahl der erneut übertragenen Byte ist in `args[0]` enthalten.
`tcpRetransSegs`	Prüfpunkt, der mit jeder Sendung eines Segments ausgelöst wird, das mindestens ein neu übertragenes Byte enthält.
`tcpRttNoUpdate`	Prüfpunkt, der immer dann ausgelöst wird, wenn Daten empfangen wurden, aber keine Zeitmarke für die RTT-Aktualisierung vorhanden war.
`tcpRttUpdate`	Prüfpunkt, der immer dann ausgelöst wird, wenn Daten einschließlich der für die RTT-Aktualisierung erforderlichen Zeitmarke empfangen wurden.
`tcpTimKeepalive`	Prüfpunkt, der bei jeder timer-basierten Keep-Alive-Verarbeitung für eine Verbindung ausgelöst wird.
`tcpTimKeepaliveDrop`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine Keep-Alive-Verarbeitung das Beenden einer Verbindung bewirkt.
`tcpTimKeepaliveProbe`	Prüfpunkt, der immer dann ausgelöst wird, wenn im Rahmen der Keep-Alive-Verarbeitung ein Keep-Alive-Probe ausgesendet wird.
`tcpTimRetrans`	Prüfpunkt, der bei jeder timer-basierten Retransmit-Verarbeitung für eine Verbindung ausgelöst wird.

TABELLE 28–8 `mib`-Prüfpunkte für TCP *(Fortsetzung)*

`tcpTimRetransDrop`	Prüfpunkt, der immer dann ausgelöst wird, wenn eine Verbindung beendet wird, da eine timer-basierte Retransmit-Verarbeitung über einen längeren Zeitraum nicht möglich war.

TABELLE 28–9 `mib`-Prüfpunkte für UDP

`udpInCksumErrs`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein Datagramm aufgrund einer ungültigen UDP-Prüfsumme verworfen wird.
`udpInDatagrams`	Prüfpunkt, der bei jedem Empfang eines UDP-Datagramms ausgelöst wird.
`udpInErrors`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein UDP-Datagramm empfangen, aber aufgrund eines fehlerhaften Paket-Headers oder einer gescheiterten Zuweisung internen Pufferplatzes verworfen wird.
`udpInOverflows`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein UDP-Datagramm empfangen, aber anschließend aufgrund von Pufferplatzmangel verworfen wird.
`udpNoPorts`	Prüfpunkt, der mit jedem Empfang eines UDP-Datagramms an einem Anschluss ausgelöst wird, der an kein Socket gebunden ist.
`udpOutDatagrams`	Prüfpunkt, der bei jeder Sendung eines UDP-Datagramms ausgelöst wird.
`udpOutErrors`	Prüfpunkt, der immer dann ausgelöst wird, wenn ein UDP-Datagramm aufgrund einer Fehlerbedingung (in der Regel wegen eines Formatfehlers des Datagramms) nicht gesendet wurde.

Argumente

Für das einzige Argument aller `mib`-Prüfpunkte gilt stets dieselbe Semantik: `args[0]` enthält den Wert, um den der Zähler erhöht werden muss. Bei den meisten `mib`-Prüfpunkten enthält `args[0]` immer den Wert `1`, bei einigen kann `args[0]` jedoch beliebige positive Werte annehmen. Für diese Prüfpunkte ist die Bedeutung von `args[0]` in der Prüfpunktbeschreibung angegeben.

Stabilität

Der Provider `mib` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Argumente	Evolving	Evolving	ISA

◆ ◆ ◆ KAPITEL 29

Der Provider `fpuinfo`

Der Provider `fpuinfo` stellt Prüfpunkte für die Simulation von Gleitkomma-Anweisungen auf SPARC-Mikroprozessoren bereit. Während die meisten Gleitkomma-Anweisungen in der Hardware ausgeführt werden, dringen andere Gleitkomma-Operationen zur Simulation in das Betriebssystem ein. Die Bedingungen, unter welchen Gleitkomma-Operationen eine Simulation durch das Betriebssystem benötigen, sind von einer Mikroprozessor-Implementierung zur nächsten unterschiedlich. Operationen, die eine Simulation erfordern, kommen selten vor. Wenn eine Anwendung derartige Operationen jedoch häufig verwendet, kann dies die Leistung stark beeinträchtigen. Der Provider `fpuinfo` ermöglicht eine schnelle Untersuchung der Gleitkomma-Simulation, die entweder über `kstat(1M)` und die Kernelstatistik `fpu_info` oder `trapstat(1M)` und die Signalfalle `fp-xcp-other` beobachtet wird.

Prüfpunkte

Der Provider `fpuinfo` stellt für jede Art simulierbarer Gleitkomma-Anweisungen einen Prüfpunkt bereit. Die Namensstabilität des Providers `fpuinfo` ist `CPU`; die Namen der Prüfpunkte sind spezifisch für die jeweilige Mikroprozessor-Implementierung und sind auf unterschiedlichen Mikroprozessoren derselben Familie möglicherweise nicht verfügbar. So sind beispielsweise einige der aufgeführten Prüfpunkte nur auf UltraSPARC-III, nicht jedoch auf UltraSPARC-III+ verfügbar, und bei anderen verhält es sich umgekehrt.

Die `fpuinfo`-Prüfpunkte sind in Tabelle 29–1 beschrieben.

TABELLE 29–1 `fpuinfo`-Prüfpunkte

`fpu_sim_fitoq`	Prüfpunkt, der mit jeder Simulation einer `fitoq`-Anweisung durch den Kernel ausgelöst wird.

TABELLE 29–1 `fpuinfo`-Prüfpunkte *(Fortsetzung)*

`fpu_sim_fitod`	Prüfpunkt, der mit jeder Simulation einer `fitod`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fitos`	Prüfpunkt, der mit jeder Simulation einer `fitos`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fxtoq`	Prüfpunkt, der mit jeder Simulation einer `fxtoq`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fxtod`	Prüfpunkt, der mit jeder Simulation einer `fxtod`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fxtos`	Prüfpunkt, der mit jeder Simulation einer `fxtos`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fqtox`	Prüfpunkt, der mit jeder Simulation einer `fqtox`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fdtox`	Prüfpunkt, der mit jeder Simulation einer `fdtox`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fstox`	Prüfpunkt, der mit jeder Simulation einer `fstox`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fqtoi`	Prüfpunkt, der mit jeder Simulation einer `fqtoi`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fdtoi`	Prüfpunkt, der mit jeder Simulation einer `fdtoi`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fstoi`	Prüfpunkt, der mit jeder Simulation einer `fstoi`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fsqrtq`	Prüfpunkt, der mit jeder Simulation einer `fsqrtq`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fsqrtd`	Prüfpunkt, der mit jeder Simulation einer `fsqrtd`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fsqrts`	Prüfpunkt, der mit jeder Simulation einer `fsqrts`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fcmpeq`	Prüfpunkt, der mit jeder Simulation einer `fcmpeq`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fcmped`	Prüfpunkt, der mit jeder Simulation einer `fcmped`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fcmpes`	Prüfpunkt, der mit jeder Simulation einer `fcmpes`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fcmpq`	Prüfpunkt, der mit jeder Simulation einer `fcmpq`-Anweisung durch den Kernel ausgelöst wird.

TABELLE 29–1 `fpuinfo`-Prüfpunkte *(Fortsetzung)*

`fpu_sim_fcmpd`	Prüfpunkt, der mit jeder Simulation einer `fcmpd`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fcmps`	Prüfpunkt, der mit jeder Simulation einer `fcmps`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fdivq`	Prüfpunkt, der mit jeder Simulation einer `fdivq`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fdivd`	Prüfpunkt, der mit jeder Simulation einer `fdivd`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fdivs`	Prüfpunkt, der mit jeder Simulation einer `fdivs`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fdmulx`	Prüfpunkt, der mit jeder Simulation einer `fdmulx`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fsmuld`	Prüfpunkt, der mit jeder Simulation einer `fsmuld`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fmulq`	Prüfpunkt, der mit jeder Simulation einer `fmulq`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fmuld`	Prüfpunkt, der mit jeder Simulation einer `fmuld`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fmuls`	Prüfpunkt, der mit jeder Simulation einer `fmuls`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fsubq`	Prüfpunkt, der mit jeder Simulation einer `fsubq`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fsubd`	Prüfpunkt, der mit jeder Simulation einer `fsubd`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fsubs`	Prüfpunkt, der mit jeder Simulation einer `fsubs`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_faddq`	Prüfpunkt, der mit jeder Simulation einer `faddq`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_faddd`	Prüfpunkt, der mit jeder Simulation einer `faddd`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fadds`	Prüfpunkt, der mit jeder Simulation einer `fadds`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fnegd`	Prüfpunkt, der mit jeder Simulation einer `fnegd`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fnegq`	Prüfpunkt, der mit jeder Simulation einer `fneqq`-Anweisung durch den Kernel ausgelöst wird.

TABELLE 29–1 `fpuinfo`-Prüfpunkte *(Fortsetzung)*

`fpu_sim_fnegs`	Prüfpunkt, der mit jeder Simulation einer `fnegs`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fabsd`	Prüfpunkt, der mit jeder Simulation einer `fabsd`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fabsq`	Prüfpunkt, der mit jeder Simulation einer `fabsq`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fabss`	Prüfpunkt, der mit jeder Simulation einer `fabss`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fmovd`	Prüfpunkt, der mit jeder Simulation einer `fmovd`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fmovq`	Prüfpunkt, der mit jeder Simulation einer `fmovq`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fmovs`	Prüfpunkt, der mit jeder Simulation einer `fmovs`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fmovr`	Prüfpunkt, der mit jeder Simulation einer `fmovr`-Anweisung durch den Kernel ausgelöst wird.
`fpu_sim_fmovcc`	Prüfpunkt, der mit jeder Simulation einer `fmovcc`-Anweisung durch den Kernel ausgelöst wird.

Argumente

Es stehen keine Argumente für `fpuinfo`-Prüfpunkte zur Verfügung.

Stabilität

Der Provider `fpuinfo` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	CPU
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	CPU

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Argumente	Evolving	Evolving	CPU

◆ ◆ ◆ KAPITEL 30

Der Provider `pid`

Der Provider `pid` ermöglicht die Ablaufverfolgung des Eintritts einer beliebigen Funktion in einem Benutzerprozess und deren Rückkehr sowie jeder mit einer absoluten Adresse oder einem Funktionsversatz angegebenen Anweisung. Wenn die Prüfpunkte nicht aktiviert sind, verursacht der Provider `pid` keine Prüftätigkeit. Aktivierte Prüfpunkte bewirken lediglich auf den überwachten Prozessen eine Prüftätigkeit.

Hinweis – Wenn der Compiler eine Funktion als Inline festlegt, wird der Prüfpunkt des Providers `pid` nicht ausgelöst. Lesen Sie in der Dokumentation Ihres Compilers nach, um das Festlegen einer Funktion als Inline zur Compilierungszeit zu vermeiden.

Hinweis – Das Verhalten des Providers `pid` kann unvorhersehbar sein, wenn er in einer Funktion, die Unterfunktionen mithilfe von Funktionszeigern aufruft, Prüfpunkte besitzt. Sie können Prüfpunkte explizit an den Adressen des·Einsprung- und Austrittspunktes einer Funktion setzen, wenn Sie solche Funktionen analysieren wollen.

Benennung von `pid`-Prüfpunkten

Eigentlich definiert der Provider `pid` eine *Klasse* von Providern. Theoretisch kann jeder Prozess einen eigenen `pid`-Provider besitzen. So würden wir beispielsweise für die Ablaufverfolgung eines Prozesses mit der ID 123 den Provider `pid123` verwenden. Bei Prüfpunkten eines dieser Provider bezieht sich der Modulteil der Prüfpunktbeschreibung auf ein in dem Adressraum des entsprechenden Prozesses geladenes Objekt. Im folgenden Beispiel wird mithilfe von `mdb(1)` eine Liste der Objekte angezeigt:

```
$ mdb -p 1234
Loading modules: [ ld.so.1 libc.so.1 ]
> ::objects
```

```
    BASE    LIMIT     SIZE NAME
   10000    34000    24000 /usr/bin/csh
ff3c0000 ff3e8000    28000 /lib/ld.so.1
ff350000 ff37a000    2a000 /lib/libcurses.so.1
ff200000 ff2be000    be000 /lib/libc.so.1
ff3a0000 ff3a2000     2000 /lib/libdl.so.1
ff320000 ff324000     4000 /platform/sun4u/lib/libc_psr.so.1
```

In der Prüfpunktbeschreibung geben Sie das Objekt nicht mit dem vollständigen Pfadnamen, sondern mit dem Dateinamen an. Außerdem können Sie das Suffix „`.1`“ oder „`so.1`“ auslassen. Alle nachfolgenden Beispiele benennen denselben Prüfpunkt:

```
pid123:libc.so.1:strcpy:entry
pid123:libc.so:strcpy:entry
pid123:libc:strcpy:entry
```

Das erste Beispiel stellt den tatsächlichen Namen des Prüfpunkts dar. Die anderen Beispiele sind praktische Aliasnamen, die intern durch den vollständigen Ladeobjektnamen ersetzt werden.

Für das Ladeobjekt der ausführbaren Datei kann der Aliasname `a.out` verwendet werden . Die folgenden beiden Prüfpunktbeschreibungen benennen denselben Prüfpunkt:

```
pid123:csh:main:return
pid123:a.out:main:return
```

Wie bei allen verankerten DTrace-Prüfpunkten gibt auch hier das Funktionsfeld der Prüfpunktbeschreibung eine Funktion im Modulfeld an. Es ist denkbar, dass im Binärcode einer Benutzeranwendung mehrere Namen für dieselbe Funktion vorhanden sind. So könnte beispielsweise `mutex_lock` ein alternativer Name für die Funktion `pthread_mutex_lock` in `libc.so.1` sein . DTrace wählt für derartige Funktionen einen kanonischen Namen und verwendet diesen intern. Das folgende Beispiel zeigt, wie DTrace Modul- und Funktionsnamen intern einer kanonischen Form neu zuordnet:

```
# dtrace -q -n pid101267:libc:mutex_lock:entry'{ \
    printf("%s:%s:%s:%s\n", probeprov, probemod, probefunc, probename); }'
pid101267:libc.so.1:pthread_mutex_lock:entry
^C
```

Diese automatische Umbenennung bedeutet, dass sich die Namen der von Ihnen aktivierten Prüfpunkte leicht von den tatsächlich aktivierten unterscheiden können. Dabei wird bei jeder Ausführung von DTrace auf Systemen mit derselben Solaris-Version derselbe kanonische Name gewählt.

Beispiele zur effektiven Verwendung des Providers `pid` finden Sie in Kapitel 33, „Ablaufverfolgung von Benutzerprozessen“.

Prüfpunkte für Funktionsgrenzen

Analog zu dem FBT-Provider, der die Ablaufverfolgung von Funktionseintritten und der Rückkehr von Funktionen im Kernel ermöglicht, bietet der Provider pid diese Fähigkeit für Benutzerprogramme. Die meisten in diesem Handbuch angeführten Beispiele für die Ablaufverfolgung von Kernel-Funktionsaufrufen mit dem FBT-Provider lassen sich in leicht abgeänderter Form auf Benutzerprozesse übertragen.

`entry`-Prüfpunkte

Ein `entry`-Prüfpunkt wird bei Aufruf der überwachten Funktion ausgelöst. Die Argumente für entry-Prüfpunkte sind die Werte der Argumente der überwachten Funktion.

`return`-Prüfpunkte

`return`-Prüfpunkte werden bei der Rückkehr von der verfolgten Funktion ausgelöst, oder wenn diese mit dem Aufruf einer anderen Funktion endet. Der Wert von `arg0` ist der Versatz der Rückgabeanweisung in der Funktion; `arg1` enthält den Rückgabewert.

Hinweis – `arg`*N* gibt ungefilterte Rohwerte vom Datentyp `int64_t` zurück. Der Provider `pid` unterstützt das Format `args[`*N*`]` nicht.

Prüfpunkte für den Funktionsversatz

Der Provider `pid` ermöglicht die Ablaufverfolgung beliebiger Anweisungen in einer Funktion. Um beispielsweise die Anweisung an 4 Byte in einer Funktion namens `main()` zu überwachen, könnten Sie einen ähnlichen Befehl wie diesen verwenden:

```
pid123:a.out:main:4
```

Jedes Mal, wenn das Programm die Anweisung an der Adresse `main+4` ausführt, wird dieser Prüfpunkt aktiviert. Die Argumente für Versatz-Prüfpunkte sind nicht definiert. Mit dem Vektor `uregs[]` lässt sich der Prozessstatus an diesen Prüfpunktstellen untersuchen. Weitere Informationen finden Sie unter „Der Vektor `uregs[]`“ auf Seite 376.

Stabilität

Der Provider `pid` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Argumente	Private	Private	Unknown

◆ ◆ ◆ KAPITEL 31

Der Provider `plockstat`

Der Provider `plockstat` stellt Prüfpunkte zum Beobachten des Verhaltens von Synchronisierungsgrundeinheiten auf Benutzerebene, einschließlich Lock-Contentions und Hold-Zeiten, bereit. Der Befehl `plockstat(1M)` ist ein DTrace-Verbraucher, der mithilfe des Providers `plockstat` Daten über Sperrereignisse auf Benutzerebene erfasst.

Überblick

Der Provider `plockstat` stellt Prüfpunkte für die folgenden Ereignisarten zur Verfügung:

Contention-Ereignisse Diese Prüfpunkte sprechen auf Konkurrenzsituationen (Contentions, Kollisionen) an einer Synchronisierungsgrundeinheit auf Benutzerebene an und werden ausgelöst, wenn ein Thread gezwungen ist, zu warten, bis eine Ressource verfügbar wird. Solaris ist im Allgemeinen für den konkurrenzfreien Betrieb (ohne Contentions) optimiert. Länger andauernde Konkurrenzsituationen sind folglich nicht zu erwarten. Diese Prüfpunkte helfen Ihnen, Fälle zu verstehen, in welchen es trotzdem zu Konkurrenz kommt. Da Konkurrenzsituationen architekturgemäß (relativ) selten sind, bewirkt die Aktivierung von contention-event-Prüfpunkten in der Regel keine starke Prüfaktivität. Sie können ohne Bedenken bezüglich einer größeren Leistungsbeeinträchtigung aktiviert werden.

Hold-Ereignisse Diese Prüfpunkte sprechen auf das Erhalten, Freigeben oder eine andere Art der Manipulation von Synchronisierungsgrundeinheiten auf Benutzerebene an. Als solche können diese Prüfpunkte helfen, beliebige Fragen über die Art und Weise der Manipulation von Synchronisierungsgrundeinheiten auf Benutzerebene zu beantworten. Da Synchronisierungsgrundeinheiten üblicherweise sehr häufig von Anwendungen erworben und freigegeben werden, kann die Aktivierung von hold-event-Prüfpunkten eine intensivere

	Prüfaktivität verursachen als contention-event-Prüfpunkte. Diese Aktivität kann zwar beträchtlich ausfallen, ist aber nicht schädlich. Die Prüfpunkte können trotzdem gefahrlos auf Produktionsanwendungen aktiviert werden.
Fehlerereignisse	Diese Prüfpunkte sprechen auf jede Art von ungewöhnlichem Verhalten beim Erwerben oder Freigeben einer Synchronisierungsgrundeinheit auf Benutzerebene an. Mithilfe dieser Ereignisse können Fehler festgestellt werden, die auftreten, wenn ein Thread durch eine Synchronisierungsgrundeinheit auf Benutzerebene blockiert wird. Fehlerereignisse sollten äußerst selten auftreten, sodass ihre Aktivierung keine ernst zu nehmende Prüfaktivität auslösen dürfte.

Mutex-Prüfpunkte

Mutexe schützen kritische Abschnitte durch gegenseitigen Ausschluss. Wenn ein Thread versucht, über `mutex_lock(3C)` oder `pthread_mutex_lock(3C)` einen von einem anderen Thread belegten Mutex zu erhalten, wird zunächst festgestellt, ob der Besitzer-Thread auf einer anderen CPU läuft. Ist dies der Fall, tritt der fordernde Thread für kurze Zeit in eine *Warteschleife* ein und wartet, bis der Mutex verfügbar wird. Wenn der Besitzer nicht auf einer anderen CPU läuft, wird der fordernde Thread *blockiert*.

Die vier `plockstat`-Prüfpunkte für Mutexe sind in Tabelle 31–1 aufgeführt. `arg0` enthält für jeden Prüfpunkt einen Zeiger auf die Struktur `mutex_t` oder `pthread_mutex_t` (identische Typen), die den Mutex darstellen.

TABELLE 31–1 Mutex-Prüfpunkte

`mutex-acquire`	Hold-event-Prüfpunkt, der unmittelbar nach dem Erlangen eines Mutex ausgelöst wird. `arg1` enthält einen booleschen Wert, der angibt, ob es sich um einen rekursiven Erwerb eines rekursiven Mutex handelt. `arg2` gibt an, wie oft der fordernde Thread die Warteschleife für diesen Mutex durchlaufen hat. `arg2` ist nur dann nicht Null, wenn der Prüfpunkt `mutex-spin` für diesen Mutex-Erwerb ausgelöst wurde.
`mutex-block`	Contention-event-Prüfpunkt, der ausgelöst wird, bevor ein Thread durch einen belegten Mutex blockiert wird. Für das Erlangen einer einzigen Sperre können sowohl `mutex-block` als auch `mutex-spin` ausgelöst werden.
`mutex-spin`	Contention-event-Prüfpunkt, der ausgelöst wird, bevor ein Thread durch einen belegten Mutex in den Wartezustand versetzt wird. Für das Erlangen einer einzigen Sperre können sowohl `mutex-block` als auch `mutex-spin` ausgelöst werden.

TABELLE 31–1 Mutex-Prüfpunkte *(Fortsetzung)*

`mutex-release`	Hold-event-Prüfpunkt, der unmittelbar nach der Freigabe eines Mutex ausgelöst wird. `arg1` enthält einen booleschen Wert, der angibt, ob es sich bei dem Ereignis um eine rekursive Freigabe eines rekursiven Mutex handelt.
`mutex-error`	Fehlerereignis-Prüfpunkt, der bei Fehlern in einer Mutex-Operation ausgelöst wird. `arg1` ist der `errno`-Wert für den aufgetretenen Fehler.

Prüfpunkte für Leser/Schreiber-Sperren

Leser/Schreiber-Sperren erlauben entweder mehreren Lesern *oder* einem einzigen Schreiber den gleichzeitigen Zugriff auf einen kritischen Abschnitt. Diese Sperren kommen in der Regel in Strukturen vor, die häufiger durchsucht als geändert werden, oder bei Threads, die viel Zeit im kritischen Abschnitt verbringen. Benutzer interagieren mit Leser/Schreiber-Sperren über die Solaris-Schnittstelle `rwlock(3C)` oder die POSIX-Schnittstelle `pthread_rwlock_init(3C)`.

Tabelle 31–2 zeigt die Prüfpunkte für Leser/Schreiber-Sperren. `arg0` enthält für jeden Prüfpunkt einen Zeiger auf die Struktur `rwlock_t` oder `pthread_rwlock_t` (identische Typen), die die adaptive Sperre darstellen. `arg1` enthält einen booleschen Wert, aus dem hervorgeht, ob es sich bei der Operation um einen Schreiber handelt.

TABELLE 31–2 Prüfpunkte für Leser/Schreiber-Sperren

`rw-acquire`	Hold-event-Prüfpunkt, der unmittelbar nach dem Erlangen einer Leser/Schreiber-Sperre ausgelöst wird.
`rw-block`	Contention-event-Prüfpunkt, der ausgelöst wird, bevor ein Thread in dem Versuch, eine Sperre zu erlangen, blockiert wird. Wenn sie aktiviert sind, werden die Prüfpunkte `rw-acquire` und `rw-error` nach `rw-block` ausgelöst.
`rw-release`	Hold-event-Prüfpunkt, der unmittelbar nach der Freigabe einer Leser/Schreiber-Sperre ausgelöst wird.
`rw-error`	Fehlerereignis-Prüfpunkt, der bei Fehlern während einer Operation mit einer Leser/Schreiber-Sperre ausgelöst wird. `arg1` ist der `errno`-Wert des aufgetretenen Fehlers.

Stabilität

Der Provider `plockstat` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA
Argumente	Evolving	Evolving	ISA

◆ ◆ ◆ KAPITEL 32

Der Provider `fasttrap`

Der Provider `fasttrap` ermöglicht die Ablaufverfolgung an spezifischen, vorprogrammierten Positionen in Benutzerprozessen. Im Gegensatz zu den meisten anderen DTrace-Providern ist der Provider `fasttrap` nicht zum Verfolgen der Systemaktivität vorgesehen, sondern soll DTrace-Verbrauchern durch Aktivieren des `fasttrap`-Prüfpunkts das Eingeben·von Informationen in das DTrace-Framework ermöglichen.

Prüfpunkte

Der Provider `fasttrap` stellt als einzigen Prüfpunkt `fasttrap:::fasttrap` zur Verfügung, der immer dann ausgelöst wird, wenn ein Prozess auf Benutzerebene einen bestimmten DTrace-Aufruf in den Kernel durchführt. Der DTrace-Aufruf zum Aktivieren des Prüfpunkts ist derzeit nicht öffentlich verfügbar.

Stabilität

Der Provider `fasttrap` beschreibt die verschiedenen Stabilitäten anhand des DTrace-Stabilitätsmechanismus gemäß der folgenden Tabelle. Weitere Informationen zum Stabilitätsmechanismus finden Sie in Kapitel 39, „Stabilität“.

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Provider	Evolving	Evolving	ISA
Modul	Private	Private	Unknown
Funktion	Private	Private	Unknown
Name	Evolving	Evolving	ISA

Element	Namensstabilität	Datenstabilität	Abhängigkeitsklasse
Argumente	Evolving	Evolving	ISA

◆ ◆ ◆ KAPITEL 33

Ablaufverfolgung von Benutzerprozessen

DTrace ist ein äußerst leistungsfähiges Tool für die Analyse des Verhaltens von Benutzerprozessen. Zum Debuggen, Analysieren von Leistungsproblemen oder einfach zum Nachvollziehen des Verhaltens komplexer Anwendungen erweist sich DTrace als unschätzbare Hilfe. Dieses Kapitel konzentriert sich auf die DTrace-Einrichtungen für die Ablaufverfolgung der Benutzerprozesstätigkeit und enthält Beispiele für deren Verwendung.

Die Subroutinen `copyin()` und `copyinstr()`

Wie DTrace mit Prozessen interagiert, unterscheidet sich geringfügig von den meisten herkömmlichen Debuggern oder Beobachtungstools. Viele dieser Hilfsmittel werden scheinbar innerhalb des Bereichs des Prozesses ausgeführt und lassen eine direkte Dereferenzierung von Zeigern auf Programmvariablen durch Benutzer zu. Anstatt scheinbar innerhalb oder als Bestandteil des Prozesses selbst, werden DTrace-Prüfpunkte im Solaris-Kernel ausgeführt. Für den Zugriff auf Prozessdaten müssen Prüfpunkte die Prozessdaten mithilfe der Subroutinen `copyin()` oder `copyinstr()` in den Adressraum des Kernels kopieren.

Betrachten wir beispielsweise den folgenden `write(2)`-Systemaufruf:

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

Das folgende D-Programm demonstriert einen falschen Versuch, den Inhalt einer dem `write(2)`-Systemaufruf übergebenen Zeichenkette auszugeben:

```
syscall::write:entry
{
    printf("%s", stringof(arg1)); /* incorrect use of arg1 */
}
```

Wenn Sie versuchen, dieses Skript auszuführen, erzeugt DTrace ähnliche Fehlermeldungen wie in diesem Beispiel:

```
dtrace: error on enabled probe ID 1 (ID 37: syscall::write:entry): \
    invalid address (0x10038a000) in action #1
```

Bei der Variable arg1, die den Wert des Parameters *Puffer* enthält, handelt es sich um eine Adresse im Speicher des Prozesses, der den Systemaufruf durchführt. Die Zeichenkette an dieser Adresse lesen wir mit der Subroutine copyinstr() und zeichnen das Ergebnis mit der Aktion printf() auf:

```
syscall::write:entry
{
    printf("%s", copyinstr(arg1)); /* correct use of arg1 */
```

Die Ausgabe dieses Skripts zeigt alle Zeichenketten, die an den write(2)-Systemaufruf übergeben werden. Gelegentlich kann jedoch eine unbrauchbare Ausgabe wie in diesem Beispiel erzeugt werden:

```
  0     37                      write:entry madaï¿½ï¿½ï¿½
```

Die Subroutine copyinstr() wirkt auf ein Eingabeargument, die Benutzeradresse einer auf Null endenden ASCII-Zeichenkette. Dem write(2)-Systemaufruf übergebene Puffer beziehen sich jedoch möglicherweise nicht auf ASCII-Zeichenketten, sondern auf Binärdaten. Damit nur so viel von der Zeichenkette ausgegeben wird, wie der Aufrufer beabsichtigt hat, verwenden Sie die Subroutine copyin(), die als zweites Argument eine Größe annimmt:

```
syscall::write:entry
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

Beachten Sie, dass DTrace den Operator stringof benötigt, um die mit copyin() abgerufenen Benutzerdaten in eine Zeichenkette umzuwandeln. Wenn Sie&;copyinstr verwenden, ist stringof() nicht erforderlich, da diese Funktion stets den Typ string zurückgibt.

Vermeiden von Fehlern

Die Subroutinen copyin() und copyinstr() können nicht aus noch „unberührten“ Benutzeradressen lesen. Das heißt, dass selbst eine gültige Adresse einen Fehler verursachen kann, wenn die Speicherseite, die diese Adresse enthält, noch nicht durch einen Seitenfehler infolge eines Zugriffs eingelagert wurde. Betrachten wir das folgende Beispiel:

```
# dtrace -n syscall::open:entry'{ trace(copyinstr(arg0)); }'
dtrace: description 'syscall::open:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
dtrace: error on enabled probe ID 2 (ID 50: syscall::open:entry): invalid address
(0x9af1b) in action #1 at DIF offset 52
```

Die Anwendung in der obigen Ausgabe funktioniert ordnungsgemäß und die Adresse in `arg0` ist gültig, bezieht sich aber auf eine Seite, auf die noch nicht durch den entsprechenden Prozess zugegriffen wurde. Um dieses Problem zu umgehen, warten Sie mit der Aufzeichnung, bis Kernel oder Anwendung die Daten verwenden. Sie könnten etwa wie im nächsten Beispiel mit der Anwendung von `copyinstr()` warten, bis der Systemaufruf zurückkehrt:

```
# dtrace -n syscall::open:entry'{ self->file = arg0; }' \
-n syscall::open:return'{ trace(copyinstr(self->file)); self->file = 0; }'
dtrace: description 'syscall::open:entry' matched 1 probe
CPU     ID                    FUNCTION:NAME
  2     51                      open:return   /dev/null
```

Ausschalten von `dtrace`(1M)-Interferenzen

Wenn Sie jeden Aufruf des `write(2)`-Systemaufrufs verfolgen, erhalten Sie eine Kette aufeinander folgender Ausgaben. Mit jedem Aufruf von `write()` ruft der Befehl `dtrace(1M)` während der Anzeige der Ausgabe `write()` auf usw. Diese Rückmeldungsschleife ist ein gutes Beispiel dafür, wie der Befehl `dtrace` in die gewünschten Daten eingreifen kann. Die Aufzeichnung dieser unerwünschten Daten lässt sich durch ein einfaches Prädikat vermeiden:

```
syscall::write:entry
/pid != $pid/
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

Die Makrovariable `$pid` wird durch die Prozess-ID des Prozesses ersetzt, der die Prüfpunkte aktiviert hat. Die Variable `pid` enthält die Prozess-ID des Prozesses, dessen Thread auf der CPU lief, auf der der Prüfpunkt ausgelöst wurde. Das Prädikat `/pid != $pid/` gewährleistet deshalb, dass das Skript keine Ereignisse in Verbindung mit der Ausführung des Skripts selbst verfolgt.

Der Provider `syscall`

Der Provider `syscall` ermöglicht die Ablaufverfolgung jedes Eintritts in einen und jeder Rückkehr von einem Systemaufruf. Systemaufrufe stellen häufig einen guten Ausgangspunkt zum Verstehen des Verhaltens eines Prozesses dar, insbesondere dann, wenn der Prozess eine lange Ausführungszeit aufweist oder lange im Kernel blockiert bleibt. Mit dem Befehl `prstat(1M)` lässt sich beobachten, wo Prozesse Zeit verbringen:

```
$ prstat -m -p 31337
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/NLWP
 13499 user1     53  44 0.0 0.0 0.0 0.0 2.5 0.0  4K  24  9K   0 mystery/6
```

Dieses Beispiel zeigt, dass der Prozess sehr viel Systemzeit verbraucht. Eine mögliche Erklärung für dieses Verhalten ist, dass der Prozess zahlreiche Systemaufrufe ausführt. Ein einfaches, in der Befehlszeile angegebenes D-Programm zeigt uns, welche Systemaufrufe am häufigsten vorkommen:

```
# dtrace -n syscall:::entry'/pid == 31337/{ @syscalls[probefunc] = count(); }'
dtrace: description 'syscall:::entry' matched 215 probes
^C

  open                                                              1
  lwp_park                                                          2
  times                                                             4
  fcntl                                                             5
  close                                                             6
  sigaction                                                         6
  read                                                             10
  ioctl                                                            14
  sigprocmask                                                     106
  write                                                          1092
```

Aus diesem Bericht geht der häufigste Systemaufruf hervor. Das ist in diesem Fall der `write(2)`-Systemaufruf. Mithilfe des Providers `syscall` lässt sich nun der Ursprung all dieser `write()`-Systemaufrufe ergründen:

```
# dtrace -n syscall::write:entry'/pid == 31337/{ @writes[arg2] = quantize(arg2); }'
dtrace: description 'syscall::write:entry' matched 1 probe
^C

           value  ------------- Distribution ------------- count
               0 |                                         0
               1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@   1037
               2 |@                                        3
               4 |                                         0
               8 |                                         0
              16 |                                         0
              32 |@                                        3
              64 |                                         0
             128 |                                         0
             256 |                                         0
             512 |                                         0
            1024 |@                                        5
            2048 |                                         0
```

Die Ausgabe zeigt, dass der Prozess viele `write()`-Systemaufrufe mit relativ geringen Datenmengen ausführt. Dieses Verhältnis könnte die Ursache des Leistungsproblems für diesen bestimmten Prozess darstellen. Dies ist ein Beispiel für eine allgemeine Methode zur Erkundung des Systemaufrufverhaltens.

Die Aktion ustack()

Die Ablaufverfolgung des Stacks eines Prozess-Threads zum Zeitpunkt der Aktivierung eines bestimmten Prüfpunkts gibt häufig sehr gründlich Aufschluss über ein Problem. Die Aktion `ustack()` dient zur Ablaufverfolgung des Stacks eines Benutzer-Threads. Wenn beispielsweise ein Prozess, der zahlreiche Dateien öffnet, gelegentlich beim open(2)-Systemaufruf scheitert, können Sie mit der Aktion `ustack()` den Codepfad ermitteln, der den fehlgeschlagenen `open()` ausführt:

```
syscall::open:entry
/pid == $1/
{
    self->path = copyinstr(arg0);
}

syscall::open:return
/self->path != NULL && arg1 == -1/
{
    printf("open for ’%s’ failed", self->path);
    ustack();
}
```

Dieses Skript veranschaulicht auch die Verwendung der Makrovariable `$1`, die den Wert des ersten in der dtrace(1M)-Befehlszeile angegebenen Operanden annimmt:

```
# dtrace -s ./badopen.d 31337
dtrace: script ’./badopen.d’ matched 2 probes
CPU     ID                    FUNCTION:NAME
  0     40                      open:return open for ’/usr/lib/foo’ failed
              libc.so.1‘__open+0x4
              libc.so.1‘open+0x6c
              420b0
              tcsh‘dosource+0xe0
              tcsh‘execute+0x978
              tcsh‘execute+0xba0
              tcsh‘process+0x50c
              tcsh‘main+0x1d54
              tcsh‘_start+0xdc
```

Die Aktion `ustack()` zeichnet Programmzählerwerte (PC) für den Stack auf und dtrace(1M) löst diese PC-Werte anhand der Symboltabellen des Prozesses in Symbolnamen auf. Kann `dtrace` einen PC-Wert nicht in ein Symbol auflösen, wird der Wert als Hexadezimalzahl ausgegeben.

Wenn ein Prozess bereits vor der Formatierung der `ustack()`-Daten für die Ausgabe vorhanden ist oder mit kill abgebrochen wird, kann `dtrace` die PC-Werte im Stack-Protokoll unter Umständen nicht in Symbolnamen umwandeln und ist gezwungen, sie als

Hexadezimalzahlen anzuzeigen. Um diese Einschränkung zu umgehen, übergeben Sie -dtrace mit der Option -c oder p gezielt einen Prozess. Ausführliche Informationen zu diesen und anderen Optionen finden Sie in Kapitel 14, „Das Dienstprogramm dtrace(1M)“. Sind Prozess-ID oder Befehl im Voraus nicht bekannt, lässt sich die Einschränkung auch mit folgendem D-Programm umgehen:

```
/*
 * This example uses the open(2) system call probe, but this technique
 * is applicable to any script using the ustack() action where the stack
 * being traced is in a process that may exit soon.
 */
 syscall::open:entry
{
    ustack();
    stop_pids[pid] = 1;
}

syscall::rexit:entry
/stop_pids[pid] != 0/
{
    printf("stopping pid %d", pid);
    stop();
    stop_pids[pid] = 0;
}
```

Das obige Skript hält einen Prozess kurz vor dessen Beendigung an, wenn die Aktion ustack() auf einen Thread in diesem Prozess angewendet wurde. Diese Technik gewährleistet, dass der Befehl dtrace in der Lage ist, die PC-Werte in symbolische Namen aufzulösen. Beachten Sie, dass der Wert von stop_pids[pid], nachdem er zum Löschen der dynamischen Variable benutzt wurde, 0 beträgt. Denken Sie daran, angehaltene Prozesse mit dem Befehl prun(1) wieder zum Laufen zu bringen. Anderenfalls sammeln sich auf dem System zahlreiche angehaltene Prozesse an.

Der Vektor uregs[]

Der Vektor uregs[] ermöglicht den Zugriff auf einzelne Benutzerregister. Die folgenden Tabellen enthalten die Indizes im Vektor uregs[] für jede unterstützte Solaris-Systemarchitektur.

TABELLE 33–1 uregs[]-Konstanten für SPARC

Konstante	Register
R_G0..R_G7	%g0....%g7 globale Register

TABELLE 33–1 uregs[]-Konstanten für SPARC *(Fortsetzung)*

Konstante	Register
R_O0..R_O7	%o0....%o7 Ausgangsregister
R_L0..R_L7	%l0....%l7 lokale Register
R_I0..R_I7	%i0....%i7 Eingangsregister
R_CCR	%ccr Bedingungscode-Register
R_PC	%pc Programmzähler
R_NPC	%npc nächster Programmzähler
R_Y	%y Multiplikations-/Divisionsregister
R_ASI	%asi Adressraum-ID-Register
R_FPRS	%fprs Status der Gleitkommaregister

TABELLE 33–2 uregs[]-Konstanten für x86

Konstante	Register
R_CS	%cs
R_GS	%gs
R_ES	%es
R_DS	%ds
R_EDI	%edi
R_ESI	%esi
R_EBP	%ebp
R_EAX	%eax
R_ESP	%esp
R_EAX	%eax
R_EBX	%ebx
R_ECX	%ecx
R_EDX	%edx
R_TRAPNO	%trapno
R_ERR	%err
R_EIP	%eip

TABELLE 33–2 uregs[]-Konstanten für x86 *(Fortsetzung)*

Konstante	Register
R_CS	%cs
R_ERR	%err
R_EFL	%efl
R_UESP	%uesp
R_SS	%ss

Auf AMD64-Plattformen hat der Vektor uregs denselben Inhalt wie auf x86-Plattformen und zusätzlich die in der folgenden Tabelle aufgeführten Elemente:

TABELLE 33–3 uregs[]-Konstanten für amd64

Konstante	Register
R_RSP	%rsp
R_RFL	%rfl
R_RIP	%rip
R_RAX	%rax
R_RCX	%rcx
R_RDX	%rdx
R_RBX	%rbx
R_RBP	%rbp
R_RSI	%rsi
R_RDI	%rdi
R_R8	%r8
R_R9	%r9
R_R10	%r10
R_R11	%r11
R_R12	%r12
R_R13	%r13
R_R14	%r14
R_R15	%r15

Die Aliasnamen in der folgenden Tabelle können auf allen Plattformen verwendet werden:

TABELLE 33–4 Gemeinsame uregs[]-Konstanten

Konstante	Register
R_PC	Programmzähler-Register
R_SP	Stack-Zeiger-Register
R_R0	erster Rückgabecode
R_R1	zweiter Rückgabecode

Der Provider pid

Der Provider pid ermöglicht die Ablaufverfolgung beliebiger Anweisungen in einem Prozess. Anders als mit den meisten anderen Providern, werden pid-Prüfpunkte nach Bedarf und gemäß den in den D-Programmen enthaltenen Prüfpunktbeschreibungen erzeugt. Deshalb sind in der Ausgabe von dtrace -l keine pid-Prüfpunkte aufgeführt, die Sie nicht selbst aktivieren.

Ablaufverfolgung von Benutzerfunktionsgrenzen

Am einfachsten kann der Provider pid als Gegenstück des Providers fbt für den Benutzerraum eingesetzt werden. Das folgende Beispielprogramm verfolgt jeden Eintritt in eine und Rückkehr aus einer einzigen Funktion. Die Makrovariable $1 (erster Operand in der Befehlszeile) ist die Prozess-ID des zu überwachenden Prozesses. Die Makrovariable $2 (zweiter Operand in der Befehlszeile) ist der Name der Funktion, aus der alle Funktionsaufrufe verfolgt werden sollen.

BEISPIEL 33–1 userfunc.d: Ablaufverfolgung von Eintritt in und Rückkehr aus einer Benutzerfunktion

```
pid$1::$2:entry
{
    self->trace = 1;
}

pid$1::$2:return
/self->trace/
{
    self->trace = 0;
}

pid$1:::entry,
pid$1:::return
```

BEISPIEL 33–1 userfunc.d: Ablaufverfolgung von Eintritt in und Rückkehr aus einer Benutzerfunktion *(Fortsetzung)*

```
/self->trace/
{
}
```

Geben Sie das obige Beispielskript ein und speichern Sie es in einer Datei namens `userfunc.d`. Machen Sie es dann mit `chmod` ausführbar. Das Skript produziert eine Ausgabe wie im folgenden Beispiel:

```
# ./userfunc.d 15032 execute
dtrace: script './userfunc.d' matched 11594 probes
  0  -> execute
  0    -> execute
  0      -> Dfix
  0      <- Dfix
  0      -> s_strsave
  0        -> malloc
  0        <- malloc
  0      <- s_strsave
  0      -> set
  0        -> malloc
  0        <- malloc
  0      <- set
  0      -> set1
  0        -> tglob
  0        <- tglob
  0      <- set1
  0      -> setq
  0        -> s_strcmp
  0        <- s_strcmp
...
```

Der Provider `pid` kann nur auf bereits laufende Prozesse angewendet werden. Mit der Makrovariable `$target` (siehe Kapitel 15, „Scripting“) und den `dtrace`-Optionen `-c` und `-p` lassen sich gewünschte Prozesse erzeugen und fassen (grab) und mit DTrace instrumentieren. Mit dem folgenden D-Skript können Sie beispielsweise die Verteilung der von einem bestimmten Subjektprozess ausgeführten `libc`-Aufrufe ermitteln:

```
pid$target:libc.so::entry
{
    @[probefunc] = count();
}
```

Um die Verteilung derartiger vom Befehl `date(1)` ausgeführter Aufrufe zu ermitteln, speichern Sie das Skript in der Datei `libc.d` und führen folgenden Befehl aus:

```
# dtrace -s libc.d -c date
dtrace: script 'libc.d' matched 2476 probes
Fri Jul 30 14:08:54 PDT 2004
dtrace: pid 109196 has exited

  pthread_rwlock_unlock                                             1
  _fflush_u                                                         1
  rwlock_lock                                                       1
  rw_write_held                                                     1
  strftime                                                          1
  _close                                                            1
  _read                                                             1
  __open                                                            1
  _open                                                             1
  strstr                                                            1
  load_zoneinfo                                                     1

...
  _ti_bind_guard                                                   47
  _ti_bind_clear                                                   94
```

Ablaufverfolgung beliebiger Anweisungen

Mit dem Provider `pid` können beliebige Anweisungen in beliebigen Benutzerfunktionen verfolgt werden. Dazu erzeugt der Provider `pid` nach Bedarf für jede Anweisung in einer Funktion einen Prüfpunkt. Der Name jedes Prüfpunkt entspricht dem Versatz der entsprechenden Anweisung in der Funktion, ausgedrückt durch eine Hexadezimalzahl. Um beispielsweise einen Prüfpunkt für die Anweisung an Versatz 0x1c in der Funktion `foo` des Moduls `bar.so` des Prozesses mit der PID 123 zu aktivieren, könnten Sie den folgenden Befehl eingeben:

```
# dtrace -n pid123:bar.so:foo:1c
```

Wenn Sie alle Prüfpunkte in der Funktion `foo`, einschließlich des Prüfpunkts für jede Anweisung, aktivieren möchten, können Sie den folgenden Befehl verwenden:

```
# dtrace -n pid123:bar.so:foo:
```

Dieser Befehl stellt eine äußerst leistungsfähige Technik zum Debuggen und Analysieren von Benutzeranwendungen dar. Seltene Fehler sind mitunter nicht leicht zu erkennen und zu beheben, da sie sich nur schwer reproduzieren lassen. Häufig kann ein Problem erst nach dem Auftreten einer Störung identifiziert werden - zu spät, um den Codepfad zu rekonstruieren. Das folgende Beispiel zeigt, wie sich durch Kombination des Providers `pid` mit spekulativer Ablaufverfolgung (siehe Kapitel 13, „Spekulative Ablaufverfolgung“) jede Anweisung in einer Funktion verfolgen und das Problem somit lösen lässt.

BEISPIEL 33–2 errorpath.d: Ablaufverfolgung des Pfads bei Fehler in Benutzerfunktionsaufruf

```
pid$1::$2:entry
{
    self->spec = speculation();
    speculate(self->spec);
    printf("%x %x %x %x %x", arg0, arg1, arg2, arg3, arg4);
}

pid$1::$2:
/self->spec/
{
    speculate(self->spec);
}

pid$1::$2:return
/self->spec && arg1 == 0/
{
    discard(self->spec);
    self->spec = 0;
}

pid$1::$2:return
/self->spec && arg1 != 0/
{
    commit(self->spec);
    self->spec = 0;
}
```

Die Ausführung von errorpath.d erzeugt eine Ausgabe wie in folgendem Beispiel:

```
# ./errorpath.d 100461 _chdir
dtrace: script ’./errorpath.d’ matched 19 probes
CPU     ID                    FUNCTION:NAME
  0  25253                     _chdir:entry 81e08 6d140 ffbfcb20 656c73 0
  0  25253                     _chdir:entry
  0  25269                         _chdir:0
  0  25270                         _chdir:4
  0  25271                         _chdir:8
  0  25272                         _chdir:c
  0  25273                        _chdir:10
  0  25274                        _chdir:14
  0  25275                        _chdir:18
  0  25276                        _chdir:1c
  0  25277                        _chdir:20
  0  25278                        _chdir:24
  0  25279                        _chdir:28
```

```
  0  25280                      _chdir:2c
  0  25268                  _chdir:return
```

◆ ◆ ◆ KAPITEL 34

Statisch definierte Ablaufverfolgung für Benutzeranwendungen

DTrace bietet Entwicklern von Benutzeranwendungen die Möglichkeit, die Fähigkeiten des Providers `pid` durch benutzerdefinierte Prüfpunkte im Anwendungscode zu ergänzen. Diese statischen Prüfpunkte bedeuten im deaktivierten Zustand so gut wie keinen Overhead und werden wie alle anderen DTrace-Prüfpunkte dynamisch aktiviert. Mit statischen Prüfpunkten können Sie DTrace-Benutzern eine Beschreibung der Anwendungssemantik bereitstellen, ohne dafür die Implementierung der Anwendungen offen zu legen oder eine Kenntnis dieser bei den Benutzern vorauszusetzen. Dieses Kapitel befasst sich mit der Definition von statischen Prüfpunkten in Benutzeranwendungen und erklärt, wie sie sich mit DTrace in Benutzerprozessen aktivieren lassen.

Auswahl der Prüfpunktstellen

Mit DTrace können Entwickler statische Prüfpunktstellen in Anwendungscode einbetten. Das gilt sowohl für vollständige Anwendungen als auch für gemeinsam genutzte Bibliotheken. Diese Prüfpunkte können in Entwicklung oder Produktion überall dort aktiviert werden, wo die Anwendung oder Bibliothek läuft. Bei der Definition von Prüfpunkten sollten Sie darauf achten, dass der Kreis Ihrer DTrace-Benutzer die semantische Bedeutung der Prüfpunkte problemlos versteht. So ließen sich beispielsweise für einen Webserver die Prüfpunkte `query-receive` und `query-respond` definieren, die sich auf die Anforderung eines Clients an den Webserver und die entsprechende Reaktion des Webservers beziehen. Diese Beispielprüfpunkte sind für die meisten DTrace-Benutzer eindeutig verständlich und beziehen sich nicht auf tief verborgene Implementierungsdetails, sondern auf Abstraktionen auf der höchsten Ebene der Anwendung. DTrace-Benutzer können diese Prüfpunkte zur Betrachtung der zeitlichen Verteilung von Anforderungen nutzen. Wenn der Prüfpunkt `query-receive` die URL-Anforderungszeichenketten als Argument wiedergibt, können DTrace-Benutzer ermitteln, welche Anforderungen die höchste Festplatten-E/A verursacht haben, indem sie diesen Prüfpunkt mit dem `io`-Provider verbinden.

Darüber hinaus ist bei der Wahl von Prüfpunktnamen und -stellen die Stabilität der beschriebenen Abstraktionen zu beachten. Bleibt dieser Prüfpunkt in künftigen Versionen der

Anwendung auch dann erhalten, wenn sich die Implementierung ändert? Ist der Prüfpunkt auf allen Systemarchitekturen sinnvoll oder bezieht er sich nur auf einen bestimmten Befehlssatz? In diesem Kapitel erfahren Sie im Detail, wie diese Entscheidungen die Definition der statischen Ablaufverfolgung beeinflussen.

Einfügen von Prüfpunkten in Anwendungen

DTrace-Prüfpunkte für Bibliotheken und ausführbare Dateien werden in einem ELF-Abschnitt des entsprechenden Anwendungscodes definiert. In diesem Teil des Handbuchs erfahren Sie, wie Sie Prüfpunkte definieren, in den Anwendungsquellcode einfügen und den Build-Prozess der Anwendung um die DTrace-Prüfpunktdefinitionen erweitern.

Definieren von Providern und Prüfpunkten

Sie definieren DTrace-Prüfpunkte in einer `.d`-Quelldatei, auf die später bei der Kompilierung und Verknüpfung der Anwendung zurückgegriffen wird. Wählen Sie zuerst einen geeigneten Namen für den Provider Ihrer Benutzeranwendung. Für jeden Prozess, der den Anwendungscode ausführt, wird die jeweilige Prozess-ID an den gewählten Providernamen angehängt. Wenn Sie beispielsweise für einen Webserver, der einen Prozess mit der ID 1203 ausführt, den Providernamen `myserv` wählen, dann lautet der DTrace-Providername für diesen Prozess `myserv1203`. Fügen Sie der `.d`-Quelldatei eine Providerdefinition wie in diesem Beispiel hinzu:

```
provider myserv {
    ...
};
```

Als Nächstes fügen Sie für jeden Prüfpunkt eine Definition und die entsprechenden Argumente hinzu. Im folgenden Beispiel werden die beiden unter „Auswahl der Prüfpunktstellen“ auf Seite 385 behandelten Prüfpunkte definiert. Der erste Prüfpunkt besitzt zwei Argumente des Typs `string`, der zweite hat kein Argument. Der D-Compiler wandelt zwei aufeinander folgende Unterstriche (`__`) in jedem Prüfpunktnamen in einen Bindestrich um (`-`).

```
provider myserv {
    probe query__receive(string, string);
    probe query__respond();
};
```

Um den Verbrauchern der Prüfpunkte Auskunft über die Wahrscheinlichkeit einer Änderung in künftigen Versionen der Anwendung zu erteilen, sollten Sie Stabilitätsattribute in die Providerdefinition aufnehmen. In Kapitel 39, „Stabilität“ finden Sie nähere Informationen zu den Stabilitätsattributen in DTrace. Das folgende Beispiel zeigt die Definition von Stabilitätsattributen:

BEISPIEL 34–1 `myserv.d`: Statisch definierte Anwendungsprüfpunkte

```
#pragma D attributes Evolving/Evolving/Common provider myserv provider
#pragma D attributes Private/Private/Unknown provider myserv module
#pragma D attributes Private/Private/Unknown provider myserv function
#pragma D attributes Evolving/Evolving/Common provider myserv name
#pragma D attributes Evolving/Evolving/Common provider myserv args

provider myserv {
    probe query__receive(string, string);
    probe query__respond();
};
```

Hinweis – D-Skripten, die nicht ganzzahlige Argumente aus benutzerdefinierten Prüfpunkten verwenden, müssen diese Argumente mithilfe der Funktionen `copyin()` und `copyinstr()` abrufen. Weitere Informationen finden Sie in Kapitel 33, „Ablaufverfolgung von Benutzerprozessen“.

Einfügen von Prüfpunkten in Anwendungscode

Nachdem Sie Ihre Prüfpunkte nun in einer `.d`-Datei definiert haben, müssen Sie den Quellcode um die Angabe der Positionen erweitern, die Ihre Prüfpunkte auslösen sollen. Betrachten wir als Beispiel den folgenden C-Anwendungsquellcode:

```
void
main_look(void)
{
    ...
    query = wait_for_new_query();
    process_query(query)
    ...
}
```

Zum Angeben der Prüfpunktstelle fügen Sie einen Verweis auf die in `<sys/sdt.h>` definierte Makro `DTRACE_PROBE()` hinzu:

```
#include <sys/sdt.h>
...

void
main_look(void)
{
    ...
    query = wait_for_new_query();
```

```
    DTRACE_PROBE2(myserv, query__receive, query->clientname, query->msg);
    process_query(query)
    ...
}
```

Der Zusatz `2` im Makronamen `DTRACE_PROBE2` bezieht sich auf die Anzahl der dem Prüfpunkt übergebenen Argumente. Die ersten beiden Argumente der Prüfpunktmakro bestehen in dem Providernamen und dem Prüfpunktnamen, die mit den D-Definitionen des Providers und des Prüfpunkts übereinstimmen müssen. Die übrigen Makro-Argumente werden den DTrace-Variablen `arg0..9` zugewiesen, wenn der Prüfpunkt ausgelöst wird. Der Quellcode Ihrer Anwendung darf mehrere Verweise auf den gleichen Provider- und Prüfpunktnamen enthalten. Sind im Quellcode mehrere Verweise auf denselben Prüfpunkt vorhanden, wird der Prüfpunkt durch jede Makroreferenz ausgelöst.

Erstellen von Anwendungen mit Prüfpunkten

Sie müssen den Build-Prozess der Anwendung um die DTrace-Provider- und -Prüfpunktdefinitionen erweitern. Bei einem typischen Erstellungsprozess werden die einzelnen Quelldateien in Objektdateien kompiliert. Die kompilierten Objektdateien werden, wie das nächste Beispiel zeigt, anschließend zu einer fertigen Anwendungsbinärdatei verknüpft:

```
cc -c src1.c
cc -c src2.c
...
cc -o myserv src1.o src2.o ...
```

Zum Aufnehmen von DTrace-Prüfpunktdefinitionen in die Anwendung fügen Sie angemessene Makefile-Regeln in den Build-Prozess ein, sodass der Befehl `dtrace` wie in folgendem Beispiel ausgeführt wird:

```
cc -c src1.c
cc -c src2.c
...
dtrace -G -32 -s myserv.d src1.o src2.o ...
cc -o myserv myserv.o src1.o src2.o ...
```

Der obige `dtrace`-Befehl führt eine Nachbearbeitung der durch die vorangehenden Compiler-Befehle generierten Objektdateien durch und erzeugt die Objektdatei `myserv.o` aus `myserv.d` sowie die anderen Objektdateien. Die `dtrace`-Option `-G` dient zum Verknüpfen der Provider- und Prüfpunktdefinitionen mit einer Benutzeranwendung. Die Option `-32` dient zum Erstellen von 32-Bit-Anwendungsbinärdateien. Die Option `-64` dient zum Erstellen von 64-Bit-Anwendungsbinärdateien.

◆ ◆ ◆ KAPITEL 35

35 Sicherheit

In diesem Kapitel werden die Zugriffsrechte beschrieben, die Systemadministratoren bestimmten Benutzern oder Prozessen für DTrace gewähren können. DTrace ermöglicht den Zugriff auf Funktionen auf Benutzerebene, Systemaufrufe, Kernelfunktionen und alle übrigen Aspekte des Systems. Die Aktionen, die mithilfe des Frameworks durchgeführt werden können, sind mitunter so wirksam, dass sie auch den Status eines Programms ändern. Ebenso wenig, wie einem Benutzer der Zugriff auf die privaten Dateien eines anderen Benutzers erlaubt werden sollte, ist es angebracht, jedem Benutzer uneingeschränkten Zugang zu sämtlichen Einrichtungen zu gewähren, die DTrace zu bieten hat. Standardmäßig kann DTrace nur vom Superuser verwendet werden. Über die Einrichtung Least Privilege ist es möglich, anderen Benutzern eine kontrollierte Verwendung von DTrace zu erlauben.

Zugriffsrechte

Solaris Least Privilege ist eine Einrichtung, über die Administratoren bestimmten Solaris-Benutzern spezifische Rechte einräumen können. Um einem Benutzer bei der Anmeldung Zugriffsrechte zuzuweisen, fügen Sie in die Datei `/etc/user_attr` eine Zeile in folgender Form ein:

Benutzername`::::defaultpriv=basic,`*Zugriffsrecht*

Um einem laufenden Prozess zusätzliche Zugriffsrechte zu gewähren, bedienen Sie sich des Befehls `ppriv(1)`.

`# ppriv -s A+`*Zugriffsrecht Prozess-ID*

Den Zugriff eines Benutzers auf die DTrace-Leistungsmerkmale regeln die drei Zugriffsrechte `dtrace_proc`, `dtrace_user` und `dtrace_kernel`. Jedes Zugriffsrecht sieht die Verwendung einer bestimmten Gruppe von DTrace-Providern, -Aktionen und -Variablen vor und entspricht einer bestimmten Verwendungsart von DTrace. Die Zugriffsmodi sind in den folgenden Abschnitten ausführlich beschrieben. Systemadministratoren sollten die Bedürfnisse

jedes Benutzers sorgfältig gegen die Auswirkungen der verschiedenen Zugriffsmodi auf Sichtbarkeit und Leistung abwägen. Ein Benutzer benötigt mindestens eines der drei DTrace-Zugriffsrechte, um DTrace nutzen zu können.

Privilegierte Verwendung von DTrace

Alle Benutzer mit einem der drei DTrace-Zugriffsrechte können Prüfpunkte des Providers `dtrace` (siehe Kapitel 17, „Der Provider `dtrace`“) aktivieren und die folgenden Aktionen und Variablen verwenden:

Provider	`dtrace`		
Aktionen	`exit`	`printf`	`tracemem`
	`discard`	`speculate`	
	`printa`	`trace`	
Variablen	`args`	`probemod`	`this`
	`epid`	`probename`	`timestamp`
	`id`	`probeprov`	`vtimestamp`
	`probefunc`	`self`	
Adressräume	Keinen		

Das Zugriffsrecht `dtrace_proc`

Das Zugriffsrecht `dtrace_proc` berechtigt zur Verwendung des·Providers `fasttrap` zur Ablaufverfolgung auf Prozessebene. Es erlaubt außerdem die Nutzung der folgenden Aktionen und Variablen:

Aktionen	`copyin`	`copyout`	`stop`
	`copyinstr`	`raise`	`ustack`
Variablen	`execname`	`pid`	`uregs`
Adressräume	Benutzer		

Dieses Zugriffsrecht gewährt keinerlei Sicht auf Solaris-Kerneldatenstrukturen oder Prozesse, für die der Benutzer keine Berechtigung besitzt.

Benutzer mit diesem Zugriffsrecht können Prüfpunkte in Prozessen erzeugen und aktivieren, die sie besitzen. Verfügt der Benutzer außerdem über das Zugriffsrecht `proc_owner`, können

Prüfpunkte in allen Prozessen erzeugt und aktiviert werden. Das Zugriffsrecht dtrace_proc ist für Benutzer vorgesehen, die am Debuggen oder der Leistungsanalyse von Benutzerprozessen beteiligt sind. Es eignet sich ideal für Entwickler, die an einer neuen Anwendung arbeiten, oder Programmierer, die sich mit der Leistungssteigerung einer Anwendung in einer Produktionsumgebung beschäftigen.

Hinweis – Benutzer mit den Zugriffsrechten dtrace_proc und proc_owner können jeden pid-Prüfpunkt aus jedem Prozess *aktivieren*, aber Prüfpunkte nur in Prozessen erzeugen, deren Zugriffsrechte eine Untergruppe der eigenen Zugriffsrechte darstellen. Ausführliche Informationen finden Sie in der Dokumentation zu Least Privilege.

Das Zugriffsrecht dtrace_proc ermöglicht eine Art des Zugriffs auf DTrace, die ausschließlich bei den Prozessen Leistungseinbußen bewirken kann, für die der Benutzer eine Berechtigung besitzt. Die instrumentierten Prozesse bedeuten eine zusätzliche Belastung der Systemressourcen und bringen insofern möglicherweise eine geringfügige Beeinträchtigung der Gesamtsystemleistung mit sich. Abgesehen von dieser Erhöhung der Gesamtlast lässt dieses Zugriffsrecht keine Instrumentation zu, die sich auf die Leistung anderer als der überwachten Prozesse auswirken würde. Da es die Benutzer zu keinen weiteren Zugriffen auf andere Prozesse oder den Kernel selbst berechtigt, empfiehlt es sich, dieses Zugriffsrecht allen Benutzern einzuräumen, die einen besseren Einblick in die inneren Abläufe ihrer eigenen Prozesse benötigen.

Das Zugriffsrecht dtrace_user

Das Zugriffsrecht dtrace_user erlaubt die leicht eingeschränkte Verwendung der Provider profile und syscall sowie die Nutzung der folgenden Aktionen und Variablen:

Provider	Profil	syscall	fasttrap
Aktionen	copyin	copyout	stop
	copyinstr	raise	ustack
Variablen	execname	pid	uregs
Adressräume	Benutzer		

Das Zugriffsrecht dtrace_user gewährt lediglich die Sicht auf Prozesse, für die der Benutzer bereits eine Berechtigung besitzt; es gewährt keinen Zugriff auf den Kernelstatus oder die Kernelaktivität. Benutzer mit diesem Zugriffsrecht haben die Möglichkeit, den Provider syscall zu aktivieren, aber die aktivierten Prüfpunkte treten nur in Prozessen in Kraft, für die der Benutzer eine Berechtigung besitzt. Ebenso kann der Provider profile zwar aktiviert

werden, aber die aktivierten Prüfpunkte treten nur in den Prozessen in Kraft, für die der Benutzer berechtigt ist, keinesfalls jedoch im Solaris-Kernel.

Die durch dieses Zugriffsrecht mögliche Instrumentation ist zwar auf bestimmte Prozesse beschränkt, kann jedoch die Gesamtsystemleistung beeinflussen. Der Provider `syscall` bewirkt eine geringfügige Leistungsbeeinträchtigung eines jeden Systemaufrufs für jeden Prozess. Indem er, ähnlich wie ein Echtzeit-Timer, in bestimmten Intervallen ausgeführt wird, lastet der Provider `profile` auf der Gesamtsystemleistung. Keine dieser Leistungseinbußen ist jedoch groß genug, als dass sie die Arbeit des Systems ernsthaft beeinträchtigen könnte. Trotzdem sollten Systemadministratoren genau abwägen, was es bedeuten kann, einem Benutzer dieses Zugriffsrecht zu erteilen. Eine Diskussion der Leistungseinbußen für die Provider `syscall` und `profile` finden Sie in Kapitel 21, „Der Provider `syscall`“ und Kapitel 19, „Der Provider `profile`“.

Das Zugriffsrecht `dtrace_kernel`

Das Zugriffsrecht `dtrace_kernel` berechtigt zur Anwendung aller Provider außer `pid` und `fasttrap` auf Prozesse, die nicht dem Benutzer gehören. Es ermöglicht außerdem den Einsatz aller Aktionen und Variablen außer kerneldestruktiven Aktionen (`breakpoint()`, `panic()`, `chill ()`). Dieses Zugriffsrecht bietet die vollständige Sicht auf den Status in Kernel und Benutzerebene. Die mit `dtrace_user` aktivierten Einrichtungen sind eine strikte Untergruppe der mit `dtrace_kernel` aktivierten.

Provider	Alle außer den oben genannten	
Aktionen	Alle außer destruktiven Aktionen	
Variablen	Alle	
Adressräume	Benutzer	Kernel

Superuser-Zugriffsrechte

Benutzer mit sämtlichen Zugriffsrechten können alle Provider und alle Aktionen einschließlich der keiner anderen Benutzerklasse erlaubten kerneldestruktiven Aktionen verwenden.

Provider	Alle
Aktionen	Alle einschließlich destruktiver Aktionen
Variablen	Alle

Adressräume	Benutzer	Kernel

◆ ◆ ◆ KAPITEL 36

Anonyme Ablaufverfolgung

Dieses Kapitel befasst sich mit der *anonymen* Ablaufverfolgung, die an keinen DTrace-Verbraucher gebunden ist. Die anonyme Ablaufverfolgung kommt dann zum Einsatz, wenn keine DTrace-Verbraucherprozesse ausgeführt werden können. Am häufigsten wird die anonyme Ablaufverfolgung von Gerätetreiberentwicklern zum Debuggen und Verfolgen der Aktivität während des Systemstarts genutzt. Jede interaktive Ablaufverfolgung kann auch anonym durchgeführt werden. Allerdings kann nur der Superuser anonyme Aktivierungen erstellen und es ist zu einem gegebenen Zeitpunkt nur jeweils eine anonyme Aktivierung möglich.

Anonyme Aktivierungen

Zum Erstellen einer anonymen Aktivierung geben Sie die Option `-A` mit einem `dtrace`(1M)-Aufruf an, der die gewünschten Prüfpunkte, Prädikate, Aktionen und Optionen enthält. `dtrace` fügt der Konfigurationsdatei des `dtrace`(7D)-Treibers (in der Regel `/kernel/drv/dtrace.conf`) eine Reihe von Treibereigenschaften hinzu, die Ihrer Anforderung entsprechen. `dtrace`(7D)-Treiber eingelesen, wenn dieser geladen wird. Der Treiber aktiviert die angegebenen Prüfpunkte mit den angegebenen Aktionen und stellt für die neue Aktivierung einen *anonymen Status* her. `dtrace`(7D)-Treiber wie alle als DTrace-Provider fungierenden Treiber nach Bedarf geladen. `dtrace`(7D)-Treiber so früh wie möglich geladen werden. `dtrace` fügt die erforderlichen `forceload`-Anweisungen für jeden benötigten DTrace-Provider sowie für `system`(4) selbst in `/etc/system` hinzu (siehe `dtrace`(7D)).

`dtrace`(7D) ausgegeben, die darauf hinweist, dass die Konfigurationsdatei erfolgreich verarbeitet wurde.

Sämtliche Optionen, einschließlich Puffergröße, Größe dynamischer Variablen, Spekulationsgröße, Anzahl der Spekulationen usw., können mit anonymer Aktivierung gesetzt werden.

Zum Löschen einer anonymen Aktivierung geben Sie `dtrace` mit der Option `-A` und ohne Prüfpunktbeschreibungen an.

Fordern des anonymen Status

Nachdem das System vollständig gebootet hat, kann durch Angabe der Option `-a` mit `dtrace` jeder anonyme Status gefordert werden. Standardmäßig bewirkt `-a`, dass der anonyme Status gefordert wird, die vorhandenen Daten verarbeitet werden und die Ausführung fortgesetzt wird. Um den anonymen Status zu nutzen (zu „verbrauchen“) und den Vorgang zu beenden, fügen Sie die Option `-e` hinzu.

Ein anonymer Status, der einmal verbraucht wurde, kann nicht mehr im Kernel ersetzt werden: Die kernelinternen Puffer, die ihn enthielten, werden zu neuen Zwecken wieder verwendet. Wenn Sie versuchen, einen nicht existierenden anonymen Ablaufverfolgungsstatus zu fordern, gibt `dtrace` eine ähnliche Meldung wie in folgendem Beispiel aus:

```
dtrace: could not enable tracing: No anonymous tracing state
```

Sollten Fehler oder Auslassungen auftreten, generiert `dtrace` bei der Forderung des anonymen Status die entsprechenden Meldungen. Die Meldungen über Fehler oder Auslassungen im anonymen oder nicht anonymen Status sind identisch.

Beispiele für die anonyme Ablaufverfolgung

Das folgende Beispiel zeigt eine anonyme DTrace-Aktivierung für jeden Prüfpunkt im Modul `iprb(7D)`:

```
# dtrace -A -m iprb
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
dtrace: run update_drv(1M) or reboot to enable changes
# reboot
```

Nach dem Neustart gibt `dtrace(7D)` auf der Konsole eine Meldung aus, die Sie darüber informiert, dass die angegebenen Prüfpunkte aktiviert werden:

```
  ...
  Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.
  Use is subject to license terms.
  NOTICE: enabling probe 0 (:iprb::)
  NOTICE: enabling probe 1 (dtrace:::ERROR)
  configuring IPv4 interfaces: iprb0.
  ...
```

Nachdem das System neu gestartet ist, kann der anonyme Status durch Angabe der Option `-a` mit `dtrace` verbraucht werden.

```
# dtrace -a
  CPU     ID                    FUNCTION:NAME
```

```
  0  22954                      _init:entry
  0  22955                     _init:return
  0  22800                   iprbprobe:entry
  0  22934          iprb_get_dev_type:entry
  0  22935         iprb_get_dev_type:return
  0  22801                  iprbprobe:return
  0  22802                  iprbattach:entry
  0  22874               iprb_getprop:entry
  0  22875              iprb_getprop:return
  0  22934          iprb_get_dev_type:entry
  0  22935         iprb_get_dev_type:return
  0  22870             iprb_self_test:entry
  0  22871            iprb_self_test:return
  0  22958            iprb_hard_reset:entry
  0  22959           iprb_hard_reset:return
  0  22862       iprb_get_eeprom_size:entry
  0  22826              iprb_shiftout:entry
  0  22828            iprb_raiseclock:entry
  0  22829           iprb_raiseclock:return
...
```

Das folgende Beispiel konzentriert sich auf die aus `iprbattach()` aufgerufenen Funktionen. Geben Sie das folgende Skript in einen Texteditor ein und speichern Sie es unter dem Namen `iprb.d`.

```
fbt::iprbattach:entry
{
    self->trace = 1;
}

fbt:::
/self->trace/
{}

fbt::iprbattach:return
{
    self->trace = 0;
}
```

Führen Sie die folgenden Befehle aus, um die vorigen Einstellungen aus der Treiberkonfigurationsdatei zu löschen, die neue Anforderung einer anonymen Ablaufverfolgung zu installieren und das System neu zu starten:

```
# dtrace -AFs iprb.d
dtrace: cleaned up old anonymous enabling in /kernel/drv/dtrace.conf
dtrace: cleaned up forceload directives in /etc/system
dtrace: saved anonymous enabling in /kernel/drv/dtrace.conf
dtrace: added forceload directives to /etc/system
```

```
dtrace: run update_drv(1M) or reboot to enable changes
# reboot
```

Nach dem Neustart gibt dtrace(7D) eine andere Meldung auf der Konsole aus, die auf die geringfügig abweichende Aktivierung hinweist:

```
...
Copyright 1983-2003 Sun Microsystems, Inc.  All rights reserved.
Use is subject to license terms.
NOTICE: enabling probe 0 (fbt::iprbattach:entry)
NOTICE: enabling probe 1 (fbt:::)
NOTICE: enabling probe 2 (fbt::iprbattach:return)
NOTICE: enabling probe 3 (dtrace:::ERROR)
configuring IPv4 interfaces: iprb0.
...
```

Wenn das System vollständig gebootet hat, führen Sie dtrace mit der Option -a und der Option -e aus. Dadurch werden die anonymen Daten verbraucht und der Vorgang wird beendet.

```
# dtrace -ae
  CPU FUNCTION
    0  -> iprbattach
    0    -> gld_mac_alloc
    0      -> kmem_zalloc
    0        -> kmem_cache_alloc
    0          -> kmem_cache_alloc_debug
    0            -> verify_and_copy_pattern
    0            <- verify_and_copy_pattern
    0            -> tsc_gethrtime
    0            <- tsc_gethrtime
    0            -> getpcstack
    0            <- getpcstack
    0            -> kmem_log_enter
    0            <- kmem_log_enter
    0          <- kmem_cache_alloc_debug
    0        <- kmem_cache_alloc
    0      <- kmem_zalloc
    0    <- gld_mac_alloc
    0    -> kmem_zalloc
    0      -> kmem_alloc
    0        -> vmem_alloc
    0          -> highbit
    0          <- highbit
    0          -> lowbit
    0          <- lowbit
    0          -> vmem_xalloc
    0            -> highbit
```

```
  0            <- highbit
  0            -> lowbit
  0            <- lowbit
  0            -> segkmem_alloc
  0              -> segkmem_xalloc
  0                -> vmem_alloc
  0                  -> highbit
  0                  <- highbit
  0                  -> lowbit
  0                  <- lowbit
  0                  -> vmem_seg_alloc
  0                    -> highbit
  0                    <- highbit
  0                    -> highbit
  0                    <- highbit
  0                    -> vmem_seg_create
...
```

◆ ◆ ◆ KAPITEL 37

Nachträgliche Ablaufverfolgung

Dieses Kapitel befasst sich mit den DTrace-Einrichtungen für die *nachträgliche* (post mortem) Extraktion und Verarbeitung von kernelinternen Daten der DTrace-Verbraucher. Im Fall eines Systemabsturzes können sich die mit DTrace aufgezeichneten Informationen als Schlüssel zur Ursache des Systemausfalls erweisen. Die DTrace-Daten können aus dem Systemspeicherabzug extrahiert und verarbeitet werden und bei der Aufklärung schwerwiegender Systemstörungen behilflich sein. Die Kombination aus diesen Post-Mortem-Fähigkeiten von DTrace und der Ringpufferrichtlinie (siehe Kapitel 11, „Puffer und Pufferung“) macht DTrace zu einer Art *Blackbox*, wie wir sie aus der kommerziellen Luftfahrt kennen, für Betriebssysteme.

Zum Extrahieren der DTrace-Daten aus einem bestimmten Speicherabzug führen Sie zunächst den Solaris Modular Debugger `mdb(1)` auf dem gewünschten Speicherabzug aus. Das MDB-Modul, das die DTrace-Funktion enthält, wird automatisch geladen. Näheres über MDB erfahren Sie in *Solaris Modular Debugger Guide* .

Anzeigen von DTrace-Verbrauchern

Wenn Sie DTrace-Daten von einem DTrace-Verbraucher extrahieren möchten, müssen Sie zunächst den gewünschten DTrace-Verbraucher angeben. Hierzu führen Sie den MDB-Befehl `::dtrace_state` aus:

```
> ::dtrace_state
    ADDR MINOR     PROC NAME                 FILE
ccaba400     2        - <anonymous>             -
ccab9d80     3 d1d6d7e0 intrstat         cda37078
cbfb56c0     4 d71377f0 dtrace           ceb51bd0
ccabb100     5 d713b0c0 lockstat         ceb51b60
d7ac97c0     6 d713b7e8 dtrace           ceb51ab8
```

Dieser Befehl gibt eine Tabelle der DTrace-Statusstrukturen aus. Jede Zeile der Tabelle enthält folgende Angaben:

- Adresse der Statusstruktur

- Unternummer des dtrace(7D)-Geräts
- Adresse der Prozessstruktur, die auf den DTrace-Verbraucher zutrifft
- Name des DTrace-Verbrauchers (oder <anonymous> bei anonymen Verbrauchern)
- Name der Dateistruktur für das geöffnete dtrace(7D)-Gerät

Um weitere Informationen über einen bestimmten DTrace-Verbraucher zu erhalten, übergeben Sie ::ps die Adresse der entsprechenden Prozessstruktur:

```
> d71377f0::ps
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R 100647 100642 100647 100638      0 0x00004008 d71377f0 dtrace
```

Anzeigen von Ablaufverfolgungsdaten

Nachdem Sie einen Verbraucher bestimmt haben, können Sie die Daten jedes unverbrauchten Puffers abrufen, indem Sie ::dtrace die Adresse der Statusstruktur übergeben. Das folgende Beispiel zeigt die Ausgabe des Befehls ::dtrace auf einer anonymen Aktivierung von syscall:::entry mit der Aktion trace(execname):

```
> ::dtrace_state
    ADDR MINOR     PROC NAME                 FILE
cbfb7a40     2        - <anonymous>             -

> cbfb7a40::dtrace
CPU     ID                    FUNCTION:NAME
  0    344               resolvepath:entry   init
  0     16                     close:entry   init
  0    202                     xstat:entry   init
  0    202                     xstat:entry   init
  0     14                      open:entry   init
  0    206                    fxstat:entry   init
  0    186                      mmap:entry   init
  0    186                      mmap:entry   init
  0    186                      mmap:entry   init
  0    190                    munmap:entry   init
  0    344               resolvepath:entry   init
  0    216                   memcntl:entry   init
  0     16                     close:entry   init
  0    202                     xstat:entry   init
  0     14                      open:entry   init
  0    206                    fxstat:entry   init
  0    186                      mmap:entry   init
  0    186                      mmap:entry   init
  0    186                      mmap:entry   init
  0    190                    munmap:entry   init
...
```

Der Befehl `::dtrace` dcmd behandelt Fehler auf dieselbe Weise wie `dtrace(1M)`. Kommt es während der Ausführung des Verbrauchers zu Auslassungen, Fehlern, spekulativen Auslassungen oder Ähnlichem gibt `::dtrace` die gleiche Meldung aus wie `dtrace(1M)`.

`::dtrace` zeigt die Ereignisse je CPU stets vom ältesten zum neuesten an. Die CPU-Puffer selbst werden in numerischer Reihenfolge wiedergegeben. Sollte es erforderlich sein, die Ereignisse von unterschiedlichen CPUs in sortierter Reihenfolge darzustellen, verfolgen Sie die Variable `timestamp`.

Durch Angabe der Option `-c` für `::dtrace` lassen sich gezielt nur die Daten einer bestimmten CPU anzeigen:

```
> cbfb7a40::dtrace -c 1
CPU     ID                    FUNCTION:NAME
  1     14                       open:entry   init
  1    206                     fxstat:entry   init
  1    186                       mmap:entry   init
  1    344                resolvepath:entry   init
  1     16                      close:entry   init
  1    202                      xstat:entry   init
  1    202                      xstat:entry   init
  1     14                       open:entry   init
  1    206                     fxstat:entry   init
  1    186                       mmap:entry   init
...
```

Beachten Sie, dass `::dtrace` nur *kernelinterne* DTrace-Daten verarbeitet. Daten, die aus dem Kernel entnommen und (durch `dtrace(1M)` oder auf andere Weise) verarbeitet wurden, können nicht mit `::dtrace` verarbeitet werden. Um zum Zeitpunkt eines Fehlers stets die größtmögliche Menge an Daten verfügbar zu haben, sollten Sie die Ringpufferrichtlinie verwenden. Weitere Informationen zu Pufferrichtlinien finden Sie in·Kapitel 11, „Puffer und Pufferung“.

Im folgenden Beispiel wird ein sehr kleiner (16 KB) Ringpuffer angelegt und alle Systemaufrufe sowie der Prozess, der sie ausführt, werden aufgezeichnet.

```
# dtrace -P syscall'{trace(curpsinfo->pr_psargs)}' -b 16k -x bufpolicy=ring
dtrace: description 'syscall:::entry' matched 214 probes
```

Wenn wir den bei der Ausführung des obigen Befehls erstellten Speicherabzug betrachten, sehen wir eine Ausgabe der Art:

```
> ::dtrace_state
    ADDR MINOR     PROC NAME                 FILE
cdccd400     3 d15e80a0 dtrace           ced065f0

> cdccd400::dtrace
```

```
CPU     ID                    FUNCTION:NAME
  0    139                   getmsg:return   mibiisa -r -p 25216
  0    138                    getmsg:entry   mibiisa -r -p 25216
  0    139                   getmsg:return   mibiisa -r -p 25216
  0    138                    getmsg:entry   mibiisa -r -p 25216
  0    139                   getmsg:return   mibiisa -r -p 25216
  0    138                    getmsg:entry   mibiisa -r -p 25216
  0    139                   getmsg:return   mibiisa -r -p 25216
  0    138                    getmsg:entry   mibiisa -r -p 25216
  0    139                   getmsg:return   mibiisa -r -p 25216
  0    138                    getmsg:entry   mibiisa -r -p 25216
  0     17                   close:return   mibiisa -r -p 25216
...
  0     96                    ioctl:entry   mibiisa -r -p 25216
  0     97                   ioctl:return   mibiisa -r -p 25216
  0     96                    ioctl:entry   mibiisa -r -p 25216
  0     97                   ioctl:return   mibiisa -r -p 25216
  0     96                    ioctl:entry   mibiisa -r -p 25216
  0     97                   ioctl:return   mibiisa -r -p 25216
  0     96                    ioctl:entry   mibiisa -r -p 25216
  0     97                   ioctl:return   mibiisa -r -p 25216
  0     16                   close:entry   mibiisa -r -p 25216
  0     17                   close:return   mibiisa -r -p 25216
  0    124                 lwp_park:entry   mibiisa -r -p 25216
  1     68                   access:entry   mdb -kw
  1     69                   access:return   mdb -kw
  1    202                     xstat:entry   mdb -kw
  1    203                    xstat:return   mdb -kw
  1     14                       open:entry   mdb -kw
  1     15                      open:return   mdb -kw
  1    206                     fxstat:entry   mdb -kw
  1    207                    fxstat:return   mdb -kw
  1    186                       mmap:entry   mdb -kw
...
  1     13                     write:return   mdb -kw
  1     10                       read:entry   mdb -kw
  1     11                      read:return   mdb -kw
  1     12                      write:entry   mdb -kw
  1     13                     write:return   mdb -kw
  1     96                      ioctl:entry   mdb -kw
  1     97                     ioctl:return   mdb -kw
  1    364                    pread64:entry   mdb -kw
  1    365                   pread64:return   mdb -kw
  1    366                   pwrite64:entry   mdb -kw
  1    367                  pwrite64:return   mdb -kw
  1    364                    pread64:entry   mdb -kw
  1    365                   pread64:return   mdb -kw
  1     38                        brk:entry   mdb -kw
```

```
  1     39                      brk:return   mdb -kw
>
```

Beachten Sie, dass die neuesten Aufzeichnungen der CPU 1 eine Reihe von `write(2)`-Systemaufrufen durch einen `mdb -kw`-Prozess enthalten. Dieses Ergebnis steht wahrscheinlich mit dem Grund für die Systemstörung in Zusammenhang, da Benutzer mit `mdb(1)` und den Optionen `-k` und `-w` Daten oder Text des laufenden Kernels modifizieren können. In diesem Fall stellen die DTrace-Daten zumindest eine interessante Spur, möglicherweise aber die Fehlerursache selbst dar.

◆ ◆ ◆ KAPITEL 38

Überlegungen zur Leistung

Aufgrund der zusätzlichen Arbeit, die DTrace im System verursacht, wirkt sich die Aktivierung von DTrace immer auf irgendeine Art und Weise auf die Systemleistung aus. Dabei handelt es sich häufig um eine vernachlässigbare Beeinträchtigung, die jedoch bei Aktivierung sehr vieler und aufwändiger Prüfpunkte beträchtlich werden kann. In diesem Kapitel werden Techniken zur Minimierung der Leistungsbeeinträchtigung durch DTrace beschrieben.

Begrenzen aktivierter Prüfpunkte

Dank dynamischer Instrumentationstechniken bietet DTrace eine unvergleichliche Reichweite der Ablaufverfolgung, die den Kernel und beliebige Benutzerprozesse einschließt. Diese Reichweite ermöglicht einerseits revolutionäre Einblicke in das Systemverhalten, kann aber andererseits auch eine sehr intensive Prüftätigkeit verursachen. Wenn zehn- oder gar hunderttausende Prüfpunkte aktiviert sind, kann es leicht zu einer beträchtlichen Auswirkung auf das System kommen. Aus diesem Grund sollten stets nur so viele Prüfpunkte aktiviert werden, wie zur Lösung eines Problems erforderlich sind. Beispielsweise empfiehlt es sich nicht, alle FBT-Prüfpunkte zu aktivieren, wenn eine etwas komplexere Aktivierung die jeweilige Fragestellung beantworten kann. So bietet es sich etwa bei einigen Fragestellungen an, sich auf ein bestimmtes Modul oder eine bestimmte Funktion zu konzentrieren.

Bei Verwendung des Providers `pid` ist besondere Vorsicht geboten. Da der Provider `pid` jede *Anweisung* instrumentieren kann, könnten Sie mehrere Millionen Prüfpunkte in einer Anwendung aktivieren und den Zielprozess dadurch extrem verlangsamen.

DTrace kann auch dann eingesetzt werden, wenn zur Beantwortung einer Frage zahlreiche Prüfpunkte aktiviert werden *müssen*. Die Aktivierung sehr vieler Prüfpunkte verlangsamt das System mitunter recht deutlich, wird das System aber keinesfalls zum Abstürzen bringen. Sie sollten also nicht zögern, viele Prüfpunkte zu aktivieren, wenn dies erforderlich ist.

Verwenden von Aggregaten

Wie bereits in Kapitel 9, „Aggregate“ besprochen, bieten DTrace-Aggregate die Möglichkeit, Daten auf skalierbare Art und Weise zusammenzufassen - zu aggregieren. Auf den ersten Blick könnte man annehmen, dass auch assoziative Vektoren eine ähnliche Funktion haben. Doch da es sich dabei um globale Allzweckvariablen handelt, lässt sich mit ihnen nicht die lineare Skalierbarkeit von Aggregaten erzielen. Wo dies möglich ist, sollten also Aggregate den assoziativen Vektoren vorgezogen werden. Das folgende Beispiel ist nicht empfehlenswert:

```
syscall::entry
{
    totals[execname]++;
}

syscall::rexit:entry
{
    printf("%40s %d\n", execname, totals[execname]);
    totals[execname] = 0;
}
```

Dieses Beispiel ist dem obigen vorzuziehen:

```
syscall::entry
{
    @totals[execname] = count();
}

END
{
    printa("%40s %@d\n", @totals);
}
```

Verwenden zwischenspeicherbarer Prädikate

DTrace-Prädikate dienen zum Herausfiltern unerwünschter Daten aus dem Experiment, indem nur Daten aufgezeichnet werden, die eine angegebene Bedingung erfüllen. Beim Aktivieren zahlreicher Prüfpunkte greift man im Allgemeinen auf Prädikate wie zum Beispiel `/self->traceme/` oder `/pid == 12345/` zurück, die einen oder mehrere spezifische Threads angeben. Obwohl viele dieser Prädikate für die meisten Threads in den meisten Prüfpunkten den Wert „falsch“ ergeben, kann die Auswertung selbst sehr aufwändig werden, wenn sie für tausende von Prüfpunkten erfolgen muss. Um diesen Aufwand herabzusetzen, speichert DTrace die Auswertung von Prädikaten, die nur thread-lokale (z. B. `/self->traceme/`) oder unveränderliche Variablen enthalten (z. B. `/pid == 12345/`) in einem Cache. Zwischengespeicherte Prädikate lassen sich unter wesentlich geringerem Aufwand auswerten als nicht zwischengespeicherte. Dies gilt insbesondere dann, wenn die Prädikate thread-lokale

Variablen, Zeichenkettenvergleiche oder andere relativ aufwändige Operationen beinhalten. Die Zwischenspeicherung von Prädikaten ist für die Benutzer transparent, unterliegt aber den in der folgenden Tabelle aufgeführten Richtlinien zum Erzeugen optimaler Prädikate:

Zwischenspeicherbar	Nicht zwischenspeicherbar
`self->mumble`	`mumble[curthread]`, `mumble[pid, tid]`
`execname`	`curpsinfo->pr_fname`, `curthread->t_procp->p_user.u_comm`
`pid`	`curpsinfo->pr_pid`, `curthread->t_procp->p_pipd->pid_id`
`tid`	`curlwpsinfo->pr_lwpid`, `curthread->t_tid`
`curthread`	`curthread->`*beliebige Komponente*, `curlwpsinfo->`*beliebige Komponente*, `curpsinfo->`*beliebige Komponente*

Das folgende Beispiel ist nicht empfehlenswert:

```
syscall::read:entry
{
    follow[pid, tid] = 1;
}

fbt:::
/follow[pid, tid]/
{}

syscall::read:return
/follow[pid, tid]/
{
    follow[pid, tid] = 0;
}
```

Das folgende Beispiel mit thread-lokalen Variablen ist dem obigen vorzuziehen:

```
syscall::read:entry
{
    self->follow = 1;
}

fbt:::
/self->follow/
{}

syscall::read:return
/self->follow/
{
```

```
    self->follow = 0;
}
```

Nur Prädikate, die *ausschließlich* aus zwischenspeicherbaren Ausdrücken bestehen, können selbst zwischengespeichert werden. Alle nachfolgenden Prädikate können zwischengespeichert werden:

```
/execname == "myprogram"/
/execname == $$1/
/pid == 12345/
/pid == $1/
/self->traceme == 1/
```

Die folgenden Beispiele mit globalen Variablen sind nicht zwischenspeicherbar:

```
/execname == one_to_watch/
/traceme[execname]/
/pid == pid_i_care_about/
/self->traceme == my_global/
```

◆ ◆ ◆ KAPITEL 39

Stabilität

Sun bietet Entwicklern häufig frühzeitigen Zugang zu neuen Technologien und Beobachtungstools, die den Benutzern einen Einblick in die internen Implementierungsdetails von Benutzer- wie Kernel-Software ermöglichen. Leider unterliegen neue Technologien und interne Implementierungsdetails naturgemäß ständigen Veränderungen aufgrund der Weiterentwicklung und Ausreifung von Schnittstellen und Implementierungen, die in Form von Aktualisierungen und Patches der Software bereitgestellt werden. Die verschiedenen Anwendungs- und Schnittstellenstabilitätsstufen dokumentiert Sun anhand der in der Manpage `attributes(5)` beschriebenen Bezeichnungen, die den Benutzern eine Vorstellung der in künftigen Versionen zu erwartenden Änderungen geben sollen.

Die beliebige Sammlung von Größen und Diensten, auf die über ein D-Programm zugegriffen werden kann, lässt sich mit einem einzigen Stabilitätsattribut nicht angemessen beschreiben. Deshalb enthalten DTrace und der D-Compiler Leistungsmerkmale, die für eine dynamische Berechnung und Beschreibung der Stabilitätsstufen der von Ihnen erstellten D-Programme sorgen. Dieses Kapitel befasst sich mit den DTrace-Leistungsmerkmalen für die Ermittlung der Programmstabilität, die Ihnen dabei helfen, stabile D-Programme zu entwerfen. Sie können die DTrace-Stabilitätsleistungsmerkmale dazu nutzen, sich über die Stabilitätsattribute Ihrer D-Programme zu informieren oder, für den Fall, dass ein Programm unerwünschte Schnittstellenabhängigkeiten aufweist, Kompilierungszeitfehler zu erzeugen.

Stabilitätsstufen

DTrace stellt zwei Arten von Stabilitätsattributen für Größen wie integrierte Variablen, Funktionen und Prüfpunkte bereit: eine *Stabilitätsstufe* und eine architektonische *Abhängigkeitsklasse*. Die DTrace-Stabilitätsstufe gibt an, wie wahrscheinlich es ist, dass eine Schnittstelle oder DTrace-Größe in einer künftigen Version oder einem Patch geändert wird, und unterstützt Sie somit bei der Risikobewertung für Skripten und Tools, die Sie auf der Grundlage von DTrace entwickeln. Die DTrace-Abhängigkeitsklasse gibt Aufschluss darüber, ob eine Schnittstelle auf alle Solaris-Plattformen und Prozessoren oder nur auf eine bestimmte

Architektur wie beispielsweise SPARC-Prozessoren zutrifft. Die zwei Attributarten zur Beschreibung von Schnittstellen können unabhängig voneinander variieren.

In der folgenden Liste sind die in DTrace verwendeten Stabilitätswerte in aufsteigender Reihenfolge, also von der niedrigsten zur höchsten Stabilität, aufgeführt. Die stabileren Schnittstellen können von allen D-Programmen und mehrschichtigen Anwendungen verwendet werden. Sun bemüht sich, zu gewährleisten, dass diese auch in künftigen Unterversionen weiterhin funktionieren. An der fehlerfreien Ausführung von Anwendungen, die nur von stabilen (Stable) Schnittstellen abhängen, sollten weder künftige Unterversionen noch Zwischen-Patches etwas ändern. Mit den weniger stabilen Schnittstellen bietet sich die Möglichkeit zum Experimentieren, Erstellen von Prototypen, Abstimmen und Debuggen auf aktuellen Systemen. Sie sollten allerdings in dem Bewusstsein eingesetzt werden, dass sie in künftigen Unterversionen inkompatibel oder gar verworfen oder aber durch Alternativen ersetzt werden könnten.

Die DTrace-Stabilitätswerte helfen Ihnen außerdem, neben der Stabilität der DTrace-Schnittstellen selbst auch die Stabilität der unter Beobachtung stehenden Softwaregrößen zu verstehen. Folglich lässt sich aus den DTrace-Stabilitätswerten auch schließen, wie wahrscheinlich es ist, dass Ihre D-Programme und mehrschichtigen Tools bei einem Upgrade oder einer Änderung des beobachteten Software-Stacks ebenfalls entsprechend geändert werden müssen.

Internal	Diese DTrace vorbehaltene Schnittstelle stellt ein Implementierungsdetail von DTrace dar. Schnittstellen des Typs „Internal“ können sich in Unter- oder Micro-Versionen ändern.
Private	Diese Sun vorbehaltene Schnittstelle wurde zur Verwendung durch andere Sun-Produkte entwickelt und noch nicht für den Gebrauch durch Kunden und ISVs öffentlich dokumentiert. Schnittstellen des Typs „Private“ können sich in Unter- oder Micro-Versionen ändern.
Obsolete	Die Schnittstelle wird in der aktuellen Version unterstützt, aber voraussichtlich in einer künftigen Unterversion entfernt. In der Regel wird die geplante Beendigung der Unterstützung einer Schnittstelle von Sun frühzeitig angekündigt. Bei einem Versuch, eine als „Obsolete“ gekennzeichnete Schnittstelle zu verwenden, gibt der D-Compiler u. U. eine Warnmeldung aus.
External	Die Schnittstelle wird von einem Drittanbieter kontrolliert, der·mit Sun nicht assoziiert ist. Sun kann im Rahmen eines Releases nach eigenem Ermessen je nach Verfügbarkeit vom Drittanbieter aktualisierte und möglicherweise inkompatible Versionen solcher Schnittstellen liefern. Sun gewährleistet in Bezug auf Schnittstellen des Typs „External“ weder eine Quell- noch Binärkompatibilität zwischen zwei beliebigen Versionen. Auf diesen Schnittstellen basierende Anwendungen sowie Patches, die externe Schnittstellen enthalten, funktionieren in künftigen Versionen möglicherweise nicht.

Unstable	Die Schnittstelle wird mit der Absicht bereitgestellt, Entwicklern einen frühzeitigen Zugang zu neuen oder schnell veränderlichen Technologien oder Implementierungsartefakten zu ermöglichen, die von zentraler Bedeutung für die Beobachtung oder das Debuggen des Systemverhaltens sind und für die in Zukunft eine stabilere Lösung geboten wird. Sun gewährleistet für Schnittstellen des Typs „Unstable“ weder Quell- noch Binärkompatibilität zwischen zwei aufeinander folgenden Unterversionen.
Evolving	Die Schnittstelle kann bald „Standard“ oder „Stable“ werden, befindet sich aber derzeit noch im Übergang. Sun bemüht sich, auch während der Fortentwicklung, eine Kompatibilität mit vorigen Versionen zu sichern. Wenn nicht aufwärtskompatible Änderungen erforderlich sind, werden diese in Unter- und Hauptversionen vorgenommen. In Micro-Versionen werden derartige Änderungen so weit wie möglich vermieden. Sollte eine solche Änderung notwendig sein, wird sie in den Versionshinweisen der betreffenden Version dokumentiert und, sofern machbar, stellt Sun Migrationshilfen bereit, die sowohl für Binärkompatibilität als auch eine unterbrechungsfreie D-Programmentwicklung sorgen.
Stable	Die Schnittstelle ist ausgereift und unterliegt der Kontrolle durch Sun. Sun bemüht sich, insbesondere in Unter- oder Micro-Versionen, nicht aufwärtskompatible Änderungen dieser Schnittstellen zu vermeiden. Falls die Unterstützung einer Schnittstelle des Typs „Stable“ beendet werden muss, kündigt Sun dies wenn möglich an, und die Stabilitätsstufe ändert sich in „Obsolete“.
Standard	Die Schnittstelle entspricht einem Industriestandard. Die Dokumentation der Schnittstelle beschreibt den Standard, den die Schnittstelle erfüllt. Standards unterstehen naturgemäß der Kontrolle einer Standardisierungsorganisation. Änderungen an der Interface sind im Einklang mit genehmigten Änderungen des jeweiligen Standards zulässig. Diese Stabilitätsstufe gelten mitunter auch für Schnittstellen, die ohne formalen Standard aus einem Industrieabkommen übernommen wurden. Unterstützung wird nur für die angegebenen Versionen eines Standards geboten; eine Unterstützung für spätere Versionen ist nicht garantiert. Genehmigt die Standardisierungsorganisation eine nicht aufwärtskompatible Änderung einer Schnittstelle des Typs „Standard“, die Sun zu unterstützen entscheidet, kündet Sun eine Strategie für Kompatibilität und Migration an.

Abhängigkeitsklassen

Angesichts der zahlreichen Betriebsplattformen und Prozessoren, die Solaris und DTrace unterstützen, kennzeichnet DTrace Schnittstellen außerdem mit einer *Abhängigkeitsklasse*, die Aufschluss darüber gibt, ob eine Schnittstelle für alle Solaris-Plattformen und Prozessoren oder nur für eine bestimmte Systemarchitektur Gültigkeit besitzt. Die Abhängigkeitsklassen stehen in keinem linearen Zusammenhang mit den bereits besprochenen Stabilitätsstufen. So kann eine DTrace-Schnittstelle beispielsweise „Stable“ sein, aber nur auf SPARC-Mikroprozessoren unterstützt werden oder „Unstable“, aber für alle Solaris-Systeme zutreffen. Die DTrace-Abhängigkeitsklassen sind in der folgenden Liste von der am wenigsten allgemeingültigen (d. h. der am meisten für eine bestimmte Architektur spezifischen) bis zur allen Architekturen gemeinsamen Schnittstelle aufgeführt.

Unknown
: Die Abhängigkeit der Schnittstelle von Architekturen ist unbekannt. DTrace kennt nicht unbedingt die Architekturabhängigkeit aller Größen wie beispielsweise von Datentypen, die in der Betriebssystemimplementierung definiert sind. Als „Unknown“ werden in der Regel Schnittstellen sehr geringer Stabilität gekennzeichnet, für die keine Abhängigkeiten berechnet werden können. Die Schnittstelle ist möglicherweise nicht verfügbar, wenn Sie DTrace auf *einer beliebigen* anderen als der derzeit verwendeten Architektur ausführen.

CPU
: Die Schnittstelle ist spezifisch auf das CPU-Modell des aktuellen Systems ausgerichtet. Mit dem Dienstprogramm `psrinfo(1M)` und der Option `-v` lassen sich die Bezeichnungen des aktuellen CPU-Modells und der Implementierung anzeigen. Schnittstellen mit CPU-Modellabhängigkeiten stehen auf anderen CPU-Implementierungen möglicherweise auch dann nicht zur Verfügung, wenn diese CPUs dieselbe Befehlssatzarchitektur (ISA) exportieren. So steht beispielsweise eine CPU-abhängige Schnittstelle auf einem UltraSPARC-III+-Mikroprozessor auf UltraSPARC-II nicht zur Verfügung, obwohl beide Prozessoren den SPARC-Befehlssatz unterstützen.

Platform
: Die Schnittstelle ist spezifisch auf die Hardware-Plattform des aktuellen Systems ausgerichtet. Eine Plattform verbindet eine Gruppe von Systemkomponenten und architektonischen Merkmalen wie zum Beispiel einen Satz unterstützter CPU-Modelle mit einem Systemnamen wie `SUNW,Ultra-Enterprise-10000` . Den aktuellen Plattformnamen rufen Sie mit `uname(1)` und der Option `-i` ab. Die Schnittstelle steht auf anderen Hardware-Plattformen möglicherweise nicht zur Verfügung.

Group
: Die Schnittstelle ist spezifisch auf die Hardware-Plattformgruppe des aktuellen Systems ausgerichtet. Eine Plattformgruppe vereint eine Menge von Plattformen mit ähnlichen Merkmalen unter einem einzigen Namen,

	wie beispielsweise `sun4u`. Den Namen der aktuellen Plattformgruppe rufen Sie mit `uname(1)` und der Option `-m` ab. Die Schnittstelle ist auf anderen Plattformen derselben Gruppe verfügbar, auf Hardware-Plattformen anderer Gruppen möglicherweise nicht.
ISA	Die Schnittstelle ist spezifisch auf die Befehlssatzarchitektur (ISA) ausgerichtet, die von den Prozessoren dieses Systems unterstützt wird. Die ISA beschreibt eine Spezifikation der auf dem Prozessor ausführbaren Software, einschließlich der Assemblersprachen-Anweisungen, Register und anderer Details. Mit dem Dienstprogramm `isainfo(1)` rufen Sie die von dem System unterstützten nativen Befehlssätze ab. Die Schnittstelle wird auf Systemen, die nur andere Befehlssätze exportieren, möglicherweise nicht unterstützt. So wird beispielsweise eine ISA-abhängige Schnittstelle auf einem Solaris SPARC-System auf einem Solaris x86-System nicht unbedingt unterstützt.
Common	Die Schnittstelle ist allen Solaris-Systemen unabhängig von der zugrunde liegenden Hardware gemeinsam. DTrace-Programme und mehrschichtige Anwendungen, die nur von Schnittstellen des Typs „Common“ abhängen, können auf anderen Solaris-Systemen mit derselben Solaris- und DTrace-Version ausgeführt und bereitgestellt werden. Die meisten DTrace-Schnittstellen sind „Common“ und können überall dort eingesetzt werden, wo Solaris läuft.

Schnittstellenattribute

DTrace beschreibt Schnittstellen anhand einer Dreiergruppe von Attributen, die aus zwei Stabilitätswerten und einer Abhängigkeitsklasse besteht. Vereinbarungsgemäß werden die Schnittstellenattribute in folgender Reihenfolge und durch Schrägstriche getrennt wiedergegeben:

Namensstabilität / Datenstabilität / Abhängigkeitsklasse

Die *Namensstabilität* einer Schnittstelle bezeichnet die Stabilitätsstufe ihres Namens, wie er im D-Programm oder in der `dtrace(1M)`-Befehlszeile erscheint. Die D-Variable `execname` zum Beispiel weist einen stabilen (Stable) Namen auf: Sun garantiert, dass dieser Name gemäß den zuvor für Schnittstellen des Typs „Stable“ aufgeführten Regeln in Ihren D-Programmen weiterhin unterstützt wird.

Die *Datenstabilität* einer Schnittstelle unterscheidet sich von der Stabilität ihres Namens. Diese Stabilitätsbezeichnung drückt das Bemühen seitens Sun aus, die von der Schnittstelle verwendeten Datenformate und die eventuell entsprechende Datensemantik beizubehalten. Die D-Variable `pid` zum Beispiel ist eine stabile (Stable) Schnittstelle: Prozess-IDs stellen in Solaris ein stabiles Konzept dar und Sun garantiert, dass die Variable `pid` im Einklang mit den

Regeln für Schnittstellen des Typs „Stable“ den Typ `pid_t` mit der auf die Prozess-ID des Threads, der einen bestimmten Prüfpunkt ausgelöst hat, gesetzten Semantik behält.

Die *Abhängigkeitsklasse* einer Schnittstelle unterscheidet sich von ihrer Namens- und Datenstabilität und gibt an, ob die Schnittstelle nur für die aktuelle Betriebsplattform oder den aktuellen Prozessor Gültigkeit besitzt.

Wie wir gleich sehen werden, kennzeichnen DTrace und der D-Compiler die Stabilitätsattribute sämtlicher DTrace-Schnittstellengrößen. Dazu gehören Provider, Prüfpunktbeschreibungen, D-Variablen, D-Funktionen, Typen und Programmanweisungen selbst. Beachten Sie, dass sich die drei Werte unabhängig voneinander verhalten. Beispielsweise besitzt die D-Variable `curthread` die Attribute Stable/Private/Common: Der Variablenname ist „Stable“ und allen Solaris-Betriebsplattformen gemeinsam („Common“), aber die Variable bietet Zugang zu einem privaten („Private“) Datenformat, das ein Produkt der Solaris-Kernelimplementierung ist. Die meisten D-Variablen besitzen die Attribute Stable/Stable/Common, so wie auch die von Ihnen selbst definierten Variablen.

Stabilitätsberechnung und -berichte

Der D-Compiler stellt für jede Prüfpunktbeschreibung und Aktionsanweisung in Ihren D-Programmen Stabilitätsberechnungen an. Mit der `dtrace`-Option `-v` erhalten Sie einen Bericht über die Stabilität eines Programms. Das folgende Beispiel zeigt ein in der Befehlszeile geschriebenes Programm:

```
# dtrace -v -n dtrace:::BEGIN'{exit(0);}'
dtrace: description 'dtrace:::BEGIN' matched 1 probe
Stability data for description dtrace:::BEGIN:
        Minimum probe description attributes
                Identifier Names: Evolving
                Data Semantics:   Evolving
                Dependency Class: Common
        Minimum probe statement attributes
                Identifier Names: Stable
                Data Semantics:   Stable
                Dependency Class: Common
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN
```

Außerdem lässt sich die `dtrace`-Option `-v` mit der Option `-e` kombinieren, wodurch dtrace das D-Programm zwar kompiliert, aber nicht ausführt und Sie die Programmstabilität ermitteln können, ohne dafür Prüfpunkte aktivieren und das Programm ausführen zu müssen. Ein weiteres Beispiel für einen Stabilitätsbericht:

```
# dtrace -ev -n dtrace:::BEGIN'{trace(curthread->t_procp);}'
Stability data for description dtrace:::BEGIN:
        Minimum probe description attributes
                Identifier Names: Evolving
                Data Semantics:   Evolving
                Dependency Class: Common
        Minimum probe statement attributes
                Identifier Names: Stable
                Data Semantics:   Private
                Dependency Class: Common
#
```

Beachten Sie, dass in dem neuen Programm die D-Variable `curthread` referenziert wird, deren Namensstabilität „Stable“, ihre Datensemantik jedoch „Private“ ist (d. h. Sie greifen auf private Implementierungsdetails des Kernels zu). Dieser Zustand wird im Stabilitätsbericht des Programms wiedergegeben. Zur Berechnung der Stabilitätsattribute im Programmbericht werden aus den entsprechenden Werten für jede Dreiergruppe an Schnittstellenattributen die niedrigste Stabilitätsstufe und Klasse ausgewählt.

Stabilitätsattribute für Prüfpunktbeschreibungen errechnen sich aus den geringsten Stabilitätsattributen aller *angegebenen* Prüfpunktbeschreibungsfelder entsprechend den vom Provider veröffentlichten Attributen. Die Attribute der verfügbaren DTrace-Provider sind in den Kapiteln über die einzelnen Provider aufgeführt. DTrace-Provider exportieren für jedes der vier Beschreibungsfelder aller von diesem Provider veröffentlichten Prüfpunkte eine Dreiergruppe von Stabilitätsattributen. Ein Providername weist deshalb u. U. eine höhere Stabilität auf als die einzelnen Prüfpunkte, die er exportiert. So hat zum Beispiel die Prüfpunktbeschreibung:

```
fbt:::
```

(sie gibt an, dass DTrace den Eintritt in und die Rückkehr aus allen Kernelfunktionen verfolgen soll) eine größere Stabilität als die Prüfpunktbeschreibung:

```
fbt:foo:bar:entry
```

die eine spezifische interne `bar()`-Funktion im Kernelmodul `foo` angibt. Der Einfachheit halber versehen die meisten Provider alle von ihnen veröffentlichten *Modul*:*Funktion*:`Name`-Werte mit nur einem Attributsatz. Außerdem geben Provider Attribute für den Vektor `args[]` bekannt, denn die Stabilität aller Prüfpunktargumente variiert von Provider zu Provider.

Sollte das Providerfeld in einer Prüfpunktbeschreibung nicht angegeben sein, werden der Beschreibung die Stabilitätsattribute Unstable/Unstable/Common zugewiesen, da die Beschreibung beim Einsatz unter einer künftigen Solaris-Version möglicherweise auf Prüfpunkte von Providern zutrifft, die bisher noch nicht existieren. Über die Zukunft des Programms im Hinblick auf Stabilität und Verhalten kann Sun keine Garantien aussprechen.

Provider sollten beim Schreiben von D-Programmklauseln stets explizit angegeben werden. Darüber hinaus werden Prüfpunktbeschreibungsfelder, die Mustervergleichszeichen (siehe Kapitel 4, „D-Programmstruktur“) oder Makrovariablen wie `$ 15` (siehe Kapitel 15, „Scripting“) enthalten, als unbestimmte Felder behandelt, da diese Beschreibungsmuster auf Provider oder Prüfpunkte zutreffen könnten, die Sun in künftigen Versionen von DTrace oder des Betriebssystems Solaris herausgeben kann.

Für die meisten D-Anweisungen errechnen sich die Stabilitätsattribute aus der niedrigsten Stabilität und Klasse der Größen in der Anweisung. Die folgenden D-Größen besitzen beispielsweise diese Attribute:

Größe	Attribute
In D integrierte Variable `curthread`	Stable/Private/Common
Benutzerdefinierte D-Variable `x`	Stable/Stable/Common

Wenn Sie die folgende D-Programmanweisung schreiben:

```
x += curthread->t_pri;
```

erhalten die Anweisungen die Attribute Stable/Private/Common, also die niedrigsten Attribute der Operanden `curthread` und `x`. Die Stabilität eines Ausdrucks errechnet sich aus den geringsten Stabilitätsattributen aller Operanden.

Allen D-Variablen, die Sie in einem Programm definieren, werden automatisch die Attribute Stable/Stable/Common zugewiesen. Außerdem werden der D-Grammatik sowie den D-Operatoren implizit die Attribute Stable/Stable/Common zugewiesen. Verweise auf Kernelsymbole anhand des Backquote-Operators (‘) stellen Artefakte der Implementierung dar und erhalten als solche stets die Attribute Private/Private/Unknown. Den in Ihrem D-Programmquellcode definierten Typen, insbesondere solchen aus Namensräumen für C- und D-Typen, werden die Attribute Stable/Stable/Common zugewiesen. Typen, die in der Betriebssystemimplementierung definiert sind und von Namensräumen für andere Typen bereitgestellt werden, erhalten die Attribute Private/Private/Unknown. Der D-Operator für die explizite Typumwandlung ergibt einen Ausdruck, dessen Stabilitätsattribute sich aus den niedrigsten Attributen des Eingangsausdrucks und den Attributen des explizit umgewandelten Ausgangstyps ableiten.

Wenn Sie mit dem C-Preprozessor C-Systemdefinitionsdateien aufnehmen, werden diese Typen dem Namensraum für C-Typen zugeordnet und erhalten die Attribute Stable/Stable/Common, da der D-Compiler davon ausgehen muss, dass Sie die Verantwortung für diese Deklarationen übernehmen. Folglich ist es denkbar, dass Sie sich im Hinblick auf die Stabilität Ihrer Programme selbst in die Irre führen, wenn Sie den C-Preprozessor verwenden, um eine Definitionsdatei mit Implementierungsartefakten aufzunehmen. Um die richtigen Stabilitätsstufen zu bestimmen, sollten Sie stets die Dokumentation zu den verwendeten Definitionsdateien zu Rate ziehen.

Erzwingen einer Stabilität

Beim Schreiben von DTrace-Skripten oder mehrschichtigen Tools kann es hilfreich sein, die spezifische Ursache von Stabilitätsproblemen zu ermitteln oder sicherzustellen, dass das Programm eine bestimmte Gruppe von Stabilitätsattributen erhält. Über die Option `dtrace -x amin=`*Attribute* können Sie den D-Compiler dazu bringen, einen Fehler zu generieren, wenn eine der Attributberechnungen eine Dreiergruppe ergibt, die geringer ist, als die von Ihnen in der Befehlszeile angegebenen Mindestwerte. Das folgende Beispiel veranschaulicht die Wirkung von `-x amin` bei einem D-Programmfragment. Beachten Sie, dass die Attribute durch drei mit `/` voneinander getrennte Bezeichnungen in der üblichen Reihenfolge wiedergegeben werden.

```
# dtrace -x amin=Evolving/Evolving/Common \
    -ev -n dtrace:::BEGIN'{trace(curthread->t_procp);}'
dtrace: invalid probe specifier dtrace:::BEGIN{trace(curthread->t_procp);}: \
    in action list: attributes for scalar curthread (Stable/Private/Common) \
    are less than predefined minimum
#
```

◆ ◆ ◆ KAPITEL 40

Übersetzer

In Kapitel 39, „Stabilität“ haben Sie erfahren, wie DTrace die Stabilitätsattribute von Programmen berechnet und meldet. Idealerweise möchten wir DTrace-Programme erstellen, die ausschließlich Schnittstellen der Stabilität „Stable“ oder „Evolving“ verbrauchen. Leider kommt man zum Debuggen von Problemen auf einer tiefen Programmebene oder beim Messen der Systemleistung manchmal nicht umhin, Prüfpunkte für interne Betriebssystemroutinen wie etwa Funktionen im Kernel zu aktivieren, anstatt Prüfpunkte für eher stabilere Schnittstellen wie beispielsweise Systemaufrufe. Bei den Daten an den tief im Software-Stack liegenden Prüfpunkten handelt es sich oft nicht um stabile Datenstrukturen wie etwa die Solaris-Systemaufrufschnittstellen, sondern um eine Sammlung von Artefakten der Implementierung. DTrace unterstützt Sie beim Schreiben stabiler D-Programme mit einer Einrichtung zum Übersetzen von Implementierungsartefakten in stabile Datenstrukturen, auf die über die D-Programmanweisungen zugegriffen werden kann.

Übersetzerdeklarationen

Ein *Übersetzer* ist eine Sammlung von D-Zuweisungsanweisungen, die der Hersteller einer Schnittstelle bereitstellt und die dazu dient, einen Eingangsausdruck in ein Objekt des Typs struct zu übersetzen. Zur Erklärung des Nutzens und der Verwendung von Übersetzern ziehen wir die in `stdio.h` definierten ANSI-C-Standardbibliotheksroutinen als Beispiel heran. Diese Routinen wirken auf eine Datenstruktur namens `FILE`, deren Implementierungsartefakte fern von C-Programmierern abstrahiert sind. Eine Standardtechnik zum Erzeugen einer Datenstrukturabstraktion besteht darin, in öffentlichen Definitionsdateien nur eine Vorausdeklaration einer Datenstruktur bereitzustellen und die zugehörige struct-Definition in einer separaten, privaten Definitionsdatei zu führen.

Wenn Sie ein C-Programm schreiben und den Dateibezeichner für eine `FILE`-Struktur kennen möchten, können Sie, anstatt eine Komponente der FILE-Struktur direkt zu dereferenzieren, mit der Funktion `fileno(3C)` den Bezeichner abrufen. Die Solaris-Definitionsdateien setzen diese Regel um, indem sie FILE als ein „undurchsichtiges“ Vordeklarations-Tag definieren, das

von C-Programmen, die <stdio.h> enthalten, nicht direkt dereferenziert werden kann. Man kann sich die Implementierung von `fileno()` innerhalb der Bibliothek `libc.so.1` in C ungefähr wie folgt vorstellen:

```
int
fileno(FILE *fp)
{
    struct file_impl *ip = (struct file_impl *)fp;

    return (ip->fd);
}
```

Unser hypothetisches `fileno()` übernimmt einen `FILE`-Zeiger als Argument, wandelt ihn in den der internen `libc`-Struktur entsprechenden Zeiger `struct file_impl` um und gibt anschließend den Wert der `fd`-Komponente der Implementierungsstruktur zurück. Weshalb implementiert Solaris Schnittstellen auf diese Weise? Indem die Details der aktuellen `libc`-Implementierung weit entfernt von den Client-Programmen abstrahiert werden, ist Sun in der Lage, so weit wie möglich eine starke Binärkompatibilität zu sichern, aber gleichzeitig die internen Implementierungsdetails von `libc` weiterzuentwickeln und zu ändern. In unserem Beispiel könnte die Komponente `fd` ihre Größe oder ihre Position innerhalb von `struct file_impl` selbst in einem Patch ändern, ohne dass bereits vorhandene Binärdateien, die `fileno(3C)` aufrufen, durch diese Änderungen beeinträchtigt würden, da sie nicht von diesen Artefakten abhängig sind.

Leider genießt jedoch Beobachtungssoftware wie DTrace nicht den Luxus, beliebige in Solaris-Bibliotheken oder im Kernel definierte C-Funktionen aufrufen zu können. Sie muss in die Implementierung blicken, um nützliche Resultate zu erbringen. Sie könnten nun zum Instrumentieren der in `stdio.h` deklarierten Routinen eine Kopie von `struct file_impl` in Ihrem D-Programm deklarieren. Dann würde das D-Programm aber von privaten Implementierungsartefakten der Bibliothek abhängen, die in einer künftigen Micro- oder Unterversion oder gar in einem Patch nicht mehr vorhanden sein könnten. Unsere Idealvorstellung ist es, ein Konstrukt für die Verwendung in D-Programmen bereitzustellen, das an die Implementierung der Bibliothek gebunden und entsprechend aktualisiert wird, aber trotzdem eine zusätzliche Abstraktionsebene mit höherer Stabilität bietet.

Zum Erstellen eines neuen Übersetzers verwenden Sie eine Deklaration der Form:

```
translator Ausgangstyp < Eingangstyp Eingangsbezeichner > {
    Komponentenname = Ausdruck ;
    Komponentenname = Ausdruck ;
    ...
};
```

Der *Ausgangstyp* ist eine Struktur, die den Ergebnistyp der Übersetzung darstellt. Der *Eingangstyp* gibt den Typ des Eingangsausdrucks an. Er steht in eckigen Klammern < > und geht einem *Eingangsbezeichner* voraus, der in den Übersetzerausdrücken als Aliasname für den

Eingangsausdruck eingesetzt werden kann. Der Rumpf des Übersetzers ist von geschweiften Klammern { } umschlossen und endet mit einem Strichpunkt (;). Er besteht aus einer Liste von *Komponentennamen* und Namen für Übersetzungsausdrücke. Jede Komponentendeklaration muss eine eindeutige Komponente des *Ausgangstyps* angeben und einem Ausdruck eines Typs zugewiesen werden, der mit dem Komponententyp kompatibel ist. Dabei gelten die Regeln für den D-Zuweisungsoperator (=).

So könnten wir beispielsweise auf Grundlage von einigen der verfügbaren `libc`-Schnittstellen eine Struktur mit stabilen Informationen über `stdio`-Dateien definieren:

```
struct file_info {
    int file_fd;   /* file descriptor from fileno(3C) */
    int file_eof;  /* eof flag from feof(3C) */
};
```

Ein hypothetischer D-Übersetzer für `FILE` in `file_info` ließe sich dann wie folgt in D deklarieren:

```
translator struct file_info < FILE *F > {
    file_fd = ((struct file_impl *)F)->fd;
    file_eof = ((struct file_impl *)F)->eof;
};
```

Der Eingangsausdruck in unserem hypothetischen Übersetzer besitzt den Typ `FILE *` und wird bezeichnet mit *Eingangsname* `F`. Der Name `F` kann dann in den Ausdrücken der Übersetzerkomponenten als eine Variable des Typs `FILE *` verwendet werden, die nur innerhalb des Rumpfs der Übersetzerdeklaration sichtbar ist. Zur Ermittlung des Werts der Ausgangskomponente `file_fd` nimmt der Übersetzer eine Typumwandlung und eine Dereferenzierung wie in der zuvor gezeigten hypothetischen Implementierung von `fileno(3C)` vor. Eine ähnliche Übersetzung erfolgt zur Ermittlung des Werts der EOF-Kennzeichnung.

Sun stellt Ihnen eine Reihe von Übersetzern für die Solaris-Schnittstellen zur Verfügung, die aus D-Programmen aufgerufen werden können. Dabei verspricht Sun, diese Übersetzer im Laufe der Weiterentwicklung der Implementierung der entsprechenden Schnittstellen im Einklang mit den zuvor definierten Regeln für die Schnittstellenstabilität zu pflegen. Bevor wir genau auf diese Übersetzer eingehen, soll geklärt werden, wie sich Übersetzer aus D aufrufen lassen. Außerdem wird die Übersetzereinrichtung selbst zur Nutzung durch Anwendungs- und Bibliotheksentwickler bereitgestellt, die eigene Übersetzer anbieten möchten, die wiederum von D-Programmierern zur Beobachtung des Status ihrer eigenen Softwarepakete verwendet werden können.

Übersetzungsoperator

Der D-Operator `xlate` dient zum Übersetzen eines Eingangsausdrucks in eine der definierten Ausgangsstrukturen. Der Operator `xlate` wird in Ausdrücken der folgenden Form verwendet:

`xlate <` *Ausgangstyp* `> (` *Eingangsausdruck* `)`

Um beispielsweise den zuvor definierten hypothetischen Übersetzer für FILE-Strukturen aufzurufen und auf die Komponente `file_fd` zuzugreifen, würden Sie folgenden Ausdruck schreiben:

```
xlate <struct file_info *>(f)->file_fd;
```

wobei `f` eine D-Variable des Typs `FILE *` ist. Dem `xlate`-Ausdruck selbst wird der durch *Ausgangstyp* definierte Typ zugewiesen. Ein fertig definierter Übersetzer ermöglicht es, nicht nur die Eingangsausdrücke in den Ausgangsstrukturtyp des Übersetzers, sondern auch in Zeiger auf diese Struktur zu übersetzen.

Wenn Sie einen Eingangsausdruck in eine Struktur übersetzen, können Sie entweder direkt mit dem Operator „`.`" eine bestimmte Komponente der Ausgabe dereferenzieren oder die gesamte übersetzte Struktur einer anderen D-Variable zuweisen, um eine Kopie der Werte aller Komponenten anzulegen. Wenn Sie eine einzelne Komponente dereferenzieren, generiert der D-Compiler nur für den Ausdruck dieser Komponente Code. Es ist nicht möglich, durch Anwendung des Operators `&` die Adresse einer übersetzten Struktur abzurufen, da das Datenobjekt selbst so lange nicht existiert, bis es kopiert oder eine seiner Komponenten referenziert wird.

Wenn Sie einen Eingangsausdruck eines Zeigers in eine Struktur übersetzen, können Sie entweder direkt mit dem Operator `->` eine bestimmte Komponente der Ausgabe dereferenzieren oder den unären Operator `*` anwenden, um den Zeiger zu dereferenzieren. In diesem Fall verhält sich das Ergebnis so, als würden Sie den Ausdruck in eine Struktur übersetzen. Wenn Sie eine einzelne Komponente dereferenzieren, generiert der D-Compiler nur für den Ausdruck dieser Komponente Code. Es ist nicht möglich, einen übersetzten Zeiger einer anderen D-Variable zuzuweisen, da das Datenobjekt selbst so lange nicht existiert, bis es entweder kopiert oder eine seiner Komponenten referenziert wird. Deshalb kann es nicht adressiert werden.

In Übersetzerdeklarationen dürfen Ausdrücke für eine oder mehr Komponenten des Ausgangstyps ausgelassen werden. Wird über einen `xlate`-Ausdruck auf eine Komponente zugegriffen, für die kein Übersetzungsausdruck definiert ist, gibt der D-Compiler eine geeignete Fehlermeldung aus und bricht die Programmkompilierung ab. Wird mithilfe einer Strukturzuweisung der gesamte Ausgangstyp kopiert, werden alle Komponenten, für die keine Übersetzungsausdrücke definiert sind, mit Nullen angefüllt.

Auf der Suche nach einem passenden Übersetzer für eine `xlate`-Operation untersucht der D-Compiler die verfügbaren Übersetzer in dieser Reihenfolge:

- Erstens: Der Compiler sucht eine Übersetzung aus dem genauen Ausdrucks-Eingangstyp in den genauen Ausgangstyp.
- Zweitens: Der Compiler *löst* die Ein- und Ausgangstypen auf, indem er alle typedef-Aliasnamen bis zu den zugrunde liegenden Typnamen verfolgt, und sucht dann eine Übersetzung aus dem aufgelösten Eingangs- in den aufgelösten Ausgangstyp.
- Drittens: Der Compiler sucht eine Übersetzung aus einem kompatiblen Eingangstyp in den aufgelösten Ausgangstyp. Dabei geht der Compiler nach denselben Regeln wie beim Ermitteln der Kompatibilität von Funktionsaufrufargumenten mit Funktionsprototypen vor und stellt fest, ob der Eingangstyp eines Ausdrucks mit dem Eingangstyp eines Übersetzers kompatibel ist.

Kann im Einklang mit diesen Regeln kein passender Übersetzer gefunden werden, gibt der D-Compiler eine entsprechende Fehlermeldung aus und die Programmkompilierung schlägt fehl.

Übersetzer für Prozessmodelle

In der DTrace-Bibliotheksdatei `/usr/lib/dtrace/procfs.d` sind verschiedene Übersetzer enthalten, die Sie in Ihren D-Programmen zum Übersetzen der Strukturen für Prozesse und Threads aus der Betriebssystem-Kernelimplementierung in die stabilen `proc(4)`-Strukturen `psinfo` und `lwpsinfo` verwenden können. Diese Strukturen kommen auch in den Dateien `/proc/`*PID*`/psinfo` und `/proc/`*PID*`/lwps/`*LWPID*`/lwpsinfo` im Solaris-Dateisystem `/proc` vor und sind in der Systemdefinitionsdatei `/usr/include/sys/procfs.h` definiert. Diese Strukturen definieren nützliche, stabile („Stable“) Informationen über Prozesse und Threads wie etwa die Prozess-ID, LWP-ID, Anfangsargumente und andere mit dem Befehl `ps(1)` abrufbare Daten. Unter `proc(4)` finden Sie eine vollständige Beschreibung der Strukturkomponenten und der Semantik.

TABELLE 40–1 `procfs.d`-Übersetzer

Eingangstyp	Attribute des Eingangstyps	Ausgangstyp	Attribute des Ausgangstyps
`proc_t *`	Private/Private/Common	`psinfo_t *`	Stable/Stable/Common
`kthread_t *`	Private/Private/Common	`lwpsinfo_t *`	Stable/Stable/Common

Stabile Übersetzungen

Übersetzer besitzen zwar die Fähigkeit, Informationen in eine stabile Datenstruktur umzuwandeln, lösen aber nicht unbedingt alle bei der Übersetzung von Daten anfallenden Stabilitätsprobleme. Verweist beispielsweise der Eingangsausdruck für eine xlate-Operation selbst auf als „Unstable“ gekennzeichnete Daten, so wird auch das entstehende D-Programm instabil, da sich die Programmstabilität stets aus der geringsten Stabilität der Menge der

D-Programmanweisungen und -ausdrücke ergibt. Aus diesem Grund ist es manchmal notwendig, einen spezifischen, stabilen Eingangsausdruck für einen Übersetzer zu definieren, sodass stabile Programme erzeugt werden können. Der inline-Mechanismus in D erleichtert die Definition solcher *stabilen Übersetzungen*.

Die DTrace-Bibliothek `procfs.d` stellt die bereits zuvor beschriebenen Variablen `curlwpsinfo` und `curpsinfo` als stabile Übersetzungen bereit. Bei der Variable `curlwpsinfo` handelt es sich zum Beispiel eigentlich um die folgende `inline`-Deklaration:

```
inline lwpsinfo_t *curlwpsinfo = xlate <lwpsinfo_t *> (curthread);
#pragma D attributes Stable/Stable/Common curlwpsinfo
```

Die Variable `curlwpsinfo` ist als inline-Übersetzung aus der Variable `curthread` (einem Zeiger auf die private Kernel-Datenstruktur, die einen Thread darstellt) in den stabilen Typ `lwpsinfo_t` definiert. Der D-Compiler verarbeitet diese Bibliotheksdatei und speichert die `inline`-Deklaration zwischen. Dadurch erscheint `curlwpsinfo` wie jede andere D-Variable. Die `#pragma`-Anweisung im Anschluss an die Deklaration setzt die Attribute des Namens `curlwpsinfo` explizit auf Stable/Stable/Common zurück, wodurch die Referenz auf `curthread` im inline-Ausdruck maskiert wird. Dank dieser Kombination aus D-Leistungsmerkmalen können D-Programmierer curthread gefahrlos als Quelle für eine Übersetzung verwenden, die im Rahmen von Änderungen in der Solaris-Implementierung von Sun aktualisiert werden kann.

◆ ◆ ◆ KAPITEL 41

Versionsverwaltung

In Kapitel 39, „Stabilität“ haben wir die DTrace-Leistungsmerkmale zum Ermitteln der Stabilitätsattribute Ihrer D-Programme kennen gelernt. Nachdem Sie ein D-Programm mit den geeigneten Stabilitätsattributen erstellt haben, möchten Sie das Programm mitunter auch an eine bestimmte *Version* der D-Programmierschnittstelle binden. Die D-Schnittstellenversion besteht in einem Bezeichner, der einem bestimmten Satz durch den D-Compiler bereitgestellter Typen, Variablen, Funktionen, Konstanten und Übersetzern anhaftet. Indem Sie eine Bindung an eine spezifische Version der D-Programmschnittstelle angeben, sorgen Sie dafür, dass Sie Ihr Programm auf künftigen Versionen von DTrace neu kompilieren können, ohne Konflikte zwischen von Ihnen definierten Programmnamen und in künftigen Versionen der D-Programmierschnittstelle definierten Namen befürchten zu müssen. Es empfiehlt sich, für jedes D-Programm, das als dauerhaftes Skript installiert (siehe Kapitel 15, „Scripting“) oder in mehrschichtigen Tools verwendet werden soll, eine Versionsbindung festzulegen.

Versionen und Versionsnummern

Gruppen von Typen, Variablen, Funktionen, Konstanten und Übersetzern einer bestimmten Softwareversion werden vom D-Compiler mit einer *Versionszeichenkette* bezeichnet. Versionszeichenketten sind Folgen durch Punkte getrennter Dezimalzahlen in der Form „*x*” (Hauptversionen), „*x.y*” (Unterversionen) oder „*x.y.z*” (Micro-Version). Für einen Versionsvergleich werden die Ziffern von links nach rechts miteinander verglichen. Stimmen die am weitesten links stehenden Zahlen nicht überein, gibt die Zeichenkette mit der höheren Zahl die höhere (und somit neuere) Version an. Stimmen die Zahlen an linker Stelle überein, fährt der Vergleich von links nach rechts mit der nächsten Zahl fort, bis das Ergebnis feststeht. Alle unbestimmten Zahlen in einer Versionszeichenkette werden bei einem Versionsvergleich als Null interpretiert.

Die DTrace-Versionszeichenketten entsprechen der in `attributes(5)` beschriebenen Sun-Standardnomenklatur für Schnittstellenversionen. Eine Änderung der D-Programmierschnittstelle schlägt sich in einer neuen Versionszeichenkette nieder. In der

folgenden Tabelle sind die von DTrace verwendeten Versionszeichenketten und die allgemeine Bedeutung der entsprechenden DTrace-Software-Version zusammengefasst.

TABELLE 41–1 DTrace-Versionen und Versionsnummern

Version	Versionsnummer	Bedeutung
Hauptversion	*x*.0	Eine Hauptversion enthält im Allgemeinen eine Reihe neuer Leistungsmerkmale; hält andere, möglicherweise inkompatible Standard-Revisionen ein; und kann - auch wenn dies unwahrscheinlich ist - geänderte Schnittstellen des Typs „Standard“ oder „Stable“ (siehe Kapitel 39, „Stabilität“) oder Ersatzschnittstellen für diese enthalten oder um solche gekürzt sein. Die erste Version der D-Programmierschnittstelle wurde als Version 1.0 bezeichnet.
Unterversion	*x.y*	Im Vergleich zu einer *x*.0 oder früheren Version (wobei y nicht Null ist) enthält eine neue Unterversion wahrscheinlich weniger neue Leistungsmerkmale, kompatible „Standard“- und „Stable“-Schnittstellen, möglicherweise inkompatible „Evolving“-Schnittstellen oder inkompatible Schnittstellen des Typs „Unstable“. Diese Änderungen können neue integrierte D-Typen, -Variablen, -Funktionen, -Konstanten und -Übersetzer umfassen. Darüber hinaus kann in einer Unterversion die Unterstützung für zuvor als „Obsolete“ gekennzeichnete Schnittstellen beendet werden (siehe Kapitel 39, „Stabilität“).
Micro-Version	*x.y.z*	Micro-Versionen sind auf Kompatibilität mit den Schnittstellen der vorigen Version (wobei z nicht Null ist) ausgerichtet und enthalten wahrscheinlich Fehlerkorrekturen, Leistungsverbesserungen und Unterstützung für zusätzliche Hardware.

Im Allgemeinen bietet jede neue Version der D-Programmierschnittstelle eine Obergruppe der Fähigkeiten der vorherigen Version abzüglich etwaiger zuvor als „Obsolete“ gekennzeichneter Schnittstellen, die dann entfernt wurden.

Optionen für die Versionsverwaltung

Standardmäßig werden alle mit `dtrace -s` kompilierten oder mit den `dtrace`-Befehlszeilenoptionen `-P`, `-m`, `-f`, `-n` oder `-i` angegebenen D-Programme an die neueste D-Programmierschnittstelle gebunden, die der D-Compiler bereitstellt. Die aktuelle Version der D-Programmierschnittstelle lässt sich mit der `dtrace`-Option `-V` ermitteln:

```
$ dtrace -V
dtrace: Sun D 1.0
$
```

Wenn Sie eine Bindung an eine bestimmte Version der D-Programmierschnittstelle festlegen möchten, können Sie die Option `version` auf eine geeignete Versionszeichenkette setzen. Ähnlich wie die DTrace-Optionen (siehe Kapitel 16, „Optionen und Tunables “) kann die Versionsoption entweder mit `dtrace -x` in der `Befehlszeile gesetzt`:

```
# dtrace -x version=1.0 -n 'BEGIN{trace("hello");}'
```

oder mithilfe der Syntax `#pragma D option` in der D-Programmquelldatei festgelegt werden:

```
#pragma D option version=1.0

BEGIN
{
    trace("hello");
}
```

Bei der Anforderung einer Versionsbindung anhand der Syntax `#pragma D option` muss diese Direktive an den Anfang der D-Programmdatei vor jede Deklaration oder Prüfpunktklausel gesetzt werden. Wenn das Argument für die Versionsbindung keiner gültigen Versionszeichenkette entspricht oder sich auf eine Version bezieht, die nicht vom D-Compiler zur Verfügung gestellt wird, erscheint eine entsprechende Fehlermeldung und die Kompilierung schlägt fehl. Folglich können Sie mithilfe der Versionsbindungseinrichtung auch bewirken, dass die Ausführung eines D-Skripts auf einer *älteren* Version von DTrace mit einer eindeutigen Fehlermeldung scheitert.

Bevor der D-Compiler die Programmdeklarationen und Klauseln kompiliert, lädt er die D-Typen, -Funktionen, -Konstanten und -Übersetzer für die entsprechende Schnittstellenversion in den eigenen Namensraum ein. Die Optionen für die Versionsbindung, die Sie angeben, bestimmen also einfach, welche Namens-, Typ- und Übersetzersätze zusätzlich zu den in Ihrem Programm definierten Variablen, Typen und Übersetzern für das Programm sichtbar sind. Durch die Versionsbindung wird verhindert, dass der D-Compiler neuere Schnittstellen einlädt, die möglicherweise mit den Deklarationen in Ihrem Programmquellcode unvereinbare Namen oder Übersetzer definieren und folglich einen Kompilierungsfehler verursachen würden. Unter „Bezeichnernamen und Schlüsselwörter“ auf Seite 49 finden Sie Tipps für die Wahl von Bezeichnernamen, die einen gewissen Schutz vor Konflikten mit Schnittstellen in künftigen Versionen von DTrace bieten.

Versionsverwaltung für Provider

Anders als Schnittstellen, die der D-Compiler zur Verfügung stellt, stehen die von DTrace bereitgestellten Provider (d. h. Prüfpunkte und Prüfpunktargumente) in keinem Zusammenhang mit der D-Programmierschnittstelle oder den oben beschriebenen Optionen für die Versionsbindung. Welche Provider-Schnittstellen zur Verfügung stehen, wird beim Laden der kompilierten Instrumentierung in die DTrace-Software im Betriebssystemkernel bestimmt. Sie hängen von der jeweiligen Befehlssatzarchitektur, der Betriebsplattform, dem Prozessor, der auf dem Solaris-System installierten Software und den aktuellen Zugriffsrechten ab. Die in Ihren D-Programmklauseln vereinbarten Prüfpunkte werden vom D-Compiler und der DTrace-Laufzeit untersucht. Wenn in Ihrem D-Programm angeforderte Prüfpunkte nicht verfügbar sind, werden entsprechende Fehlermeldungen ausgegeben. Diese

Leistungsmerkmale stehen in keinem linearen Zusammenhang mit der Version der D-Programmierschnittstelle, da DTrace-Provider keine Schnittstellen exportieren, die mit Definitionen in Ihren D-Programmen in Konflikt treten können. Das heißt, Sie können Prüfpunkte in D nur aktivieren, nicht aber definieren, und Prüfpunktnamen werden getrennt von anderen D-Programmbezeichnern in anderen Namensräumen geführt.

DTrace-Provider werden mit einer bestimmten Version von Solaris geliefert und in der entsprechenden Version des Handbuchs zur dynamischen Ablaufverfolgung in Solaris beschrieben. In den Kapiteln dieses Handbuchs, die sich mit den einzelnen Providern befassen, werden außerdem relevante Änderungen des jeweiligen Providers oder neue Leistungsmerkmale erläutert, die er zur Verfügung stellt. Mit der `dtrace`-Option `-l` können Sie den Satz der auf einem Solaris-System verfügbaren Provider und Prüfpunkte prüfen. Die Schnittstellen eines Providers sind durch die DTrace-Stabilitätsattribute bezeichnet, sodass Sie mithilfe der DTrace-Leistungsmerkmale für die Ausgabe eines Stabilitätsberichts (siehe Kapitel 39, „Stabilität“) feststellen können, wie wahrscheinlich es ist, dass die Provider-Schnittstellen, die in Ihrem D-Programm zum Einsatz kommen, in künftigen Solaris-Versionen verändert oder überhaupt nicht mehr enthalten sein werden.

Glossar

Aggregat Ein Objekt, in dem das Ergebnis einer *Aggregatfunktion* gemäß der formellen Definition in Kapitel 9, „Aggregate“ gespeichert wird und das durch ein Tupel von Ausdrücken indiziert wird, über die sich die Ergebnisse organisieren lassen.

Aktion Ein über das DTrace-Framework implementiertes Verhalten bei der Auslösung von Prüfpunkten, das entweder die Ablaufverfolgung von Daten oder die Änderung des DTrace-externen Systemstatus bewirkt. Bei Aktionen kann es sich u. a. um die Ablaufverfolgung von Daten, das Anhalten von Prozessen oder das Erfassen von Stack-Protokollen handeln.

Aktivierung Eine Gruppe aktivierter Prüfpunkte mit den zugehörigen Prädikaten und Aktionen.

DTrace Eine Einrichtung für die dynamische Ablaufverfolgung (auch Tracing genannt), die prägnante Antworten auf beliebige Fragen liefert.

Klausel Eine D-Programmdeklaration, die aus einer Liste von Prüfpunktangaben, einem optionalen Prädikat und einer optionalen Liste mit Aktionsanweisungen besteht und von geschweiften Klammern `{ }` umschlossen ist.

Prädikat Ein logischer Ausdruck, der bestimmt, ob eine Gruppe von Ablaufverfolgungsaktionen bei der Auslösung eines Prüfpunkts ausgeführt werden soll. Jeder D-Programmklausel kann ein von Schrägstrichen `/ /` umschlossenes Prädikat zugeordnet werden.

Provider Ein Kernelmodul, das einen bestimmten Instrumentationstyp für das DTrace-Framework implementiert. Der Provider exportiert einen Prüfpunkt-Namensraum sowie eine Stabilitätsmatrix für seine Namens- und Datensemantik (siehe die entsprechenden Kapitel in diesem Buch).

Prüfpunkt Eine Stelle oder Aktivität im System, an die DTrace dynamisch eine Instrumentation, einschließlich Prädikat und Aktionen, binden kann. Jeder Prüfpunkt wird durch ein aus der Angabe von Provider, Modul, Funktion und semantischem Namen bestehendes Tupel bezeichnet. Ein Prüfpunkt kann entweder mit einem bestimmten Modul und einer Funktion *verankert* oder, wenn er sich auf keine bestimmte Stelle im Programm bezieht (z. B. einen `profile`-Timer), *nicht verankert* sein.

Subroutine Ein über das DTrace-Framework implementiertes Verhalten bei der Auslösung von Prüfpunkten, das den internen DTrace-Status ändert, aber keine Daten verfolgt. Wie Aktionen werden auch Subroutinen mit der Syntax für D-Funktionsaufrufe angefordert.

Übersetzer Eine Sammlung von D-Zuweisungsanweisungen, die Implementierungsdetails eines bestimmten instrumentierten Subsystems in ein Objekt des Typs `struct` umwandeln, das eine Schnittstelle von höherer Stabilität bildet, als der Eingangsausdruck.

Verbraucher Ein Programm, das DTrace zum Aktivieren der Instrumentation nutzt und den entstehenden Stream von Ablaufverfolgungsdaten ausliest. Der Befehl `dtrace` ist der kanonische DTrace-Verbraucher. Einen weiteren, spezialisierten DTrace-Verbraucher stellt das Dienstprogramm `lockstat(1M)` dar.

Index

B

C

D

E

F

Q

R

S

T

U

V

W

Z