Solaris 動的トレースガイド

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

Part No: 819–0395–12
2008 年 10 月

原典: Solaris Dynamic Tracing Guide

Part No: 817-6223-12

Revision A

081107@21288

目次

図目次

表目次

例目次

はじめに

DTrace は、Solaris™ オペレーティングシステムで使用する包括的な動的トレースフレームワークです。 管理者、開発者、サービス担当者は、DTrace の強力なインフラストラクチャを利用して、オペレーティングシステムやユーザープログラムの動作に関するさまざまな問題に簡潔に答えることができます。『Solaris 動的トレースガイド』では、DTrace を使ってシステムの動作を監視、デバッグ、チューニングする方法について説明します。本書には、バンドルされている DTrace 監視ツールと D プログラミング言語のリファレンスも含まれています。

注 – このリリースでは、SPARC® および x86 系列のプロセッサアーキテクチャー (UltraSPARC®、SPARC64、AMD64、Pentium、Xeon EM64T) を使用するシステムをサポートします。サポートされるシステムについては、Solaris 10 Hardware Compatibility List (http://www.sun.com/bigadmin/hcl/) を参照してください。本書では、プラットフォームにより実装が異なる場合は、それを特記します。

本書では、「x86」という用語は AMD 64 あるいは Intel Xeon/Pentium 製品系列と互換性のあるプロセッサを使用して製造された 32 ビットおよび 64 ビットシステムを意味します。サポートされるシステムについては、Solaris 10 Hardware Compatibility List を参照してください。

対象読者

DTrace は、システムの動作を把握したいと考えているユーザーにとって理想的なツールです。DTrace は、Solaris に組み込まれた包括的な動的トレース機能です。DTrace の機能を利用して、ユーザープログラムの動作を検査できます。また、DTrace の機能を利用して、オペレーティングシステムの動作を検査することもできます。DTrace は、システム管理者やアプリケーション開発者によって使用される機能であり、本稼働システム上での使用に適しています。DTrace には、システムの動作を調べる機能、複数のソフトウェア層でパフォーマンス上の問題を検出する機能、システムの異常な動作の原因を突き止める機能などがあります。これから見ていくように、DTrace では、システムを動的に計測し、DTrace D プログラミング言語で記述して任意の問題に対して簡潔な答えを迅速に導き出すような、独自のカスタムプログラムを作成することもできます。

Solaris ユーザーは、DTrace を使って、次の処理を実行できます。

- 多数のプローブを動的に有効化し、管理する
- 論理述語とアクションをプローブに動的に関連付ける
- トレースバッファーとバッファーポリシーを動的に管理する
- 稼働中のシステムまたはクラッシュダンプのトレースデータを表示し、検査する

Solaris の開発者や管理者は、DTrace を使って、次の処理を実行できます。

- DTrace 機能を使ったカスタムスクリプトを実装する
- DTrace を使ってトレースデータを取得する階層化ツールを実装する

本書では、DTrace の使用に際して必要なあらゆる情報を提供します。C 言語のようなプログラミング言語、`awk(1)`、`perl(1)` のようなスクリプト言語の基礎知識は、DTrace や D プログラミング言語について学習する上で役立ちます。しかし、こうしたプログラミング言語、スクリプト言語に関する専門知識は必須ではありません。プログラミングやスクリプト作成の経験がまったくないユーザーは、23 ページの「関連情報」で紹介する参考書籍をお読みになることをお勧めします。

内容の紹介

第 1 章「はじめに」では、DTrace の機能の概要を示し、D プログラミング言語の基本情報を提供します。第 2 章「型、演算子、および式」、第 3 章「変数」、および第 4 章「D プログラムの構造」では、D の基本概念についてより詳しく説明し、D プログラムを動的計測プログラムとして使用する方法について説明します。これらの章は、すべてのユーザーを対象としています。

第 5 章「ポインタと配列」、第 6 章「文字列」、第 7 章「構造体と共用体」、および第 8 章「型と定数の定義」では、D 言語のその他の機能について説明します。そのほとんどは、C、C++、および Java™ プログラマであればすでに知っている内容です。これらの章は、こうしたプログラミング言語に関する予備知識のないユーザーを対象としています。プログラミング上級者は、これらの章を飛ばして、次の章に進んでもかまいません。

第 9 章「集積体」と第 10 章「アクションとサブルーチン」では、DTrace の強力なデータ「集積」のための要素と、トレースに使用できる一連の組み込みアクションについて説明します。これらの章は、すべてのユーザーを対象としています。

第 11 章「バッファーとバッファリング」では、データのバッファーに関する DTrace ポリシーと、こうしたポリシーの構成方法について説明します。この章は、D プログラムの構築と実行に精通してからお読みください。

第 12 章「出力書式」では、D の出力の書式設定アクションと、トレースデータの書式設定に関するデフォルトポリシーについて説明します。この章は、C の `printf()` 関数についての知識があれば、ざっと読むだけでかまいません。`printf()` についてまったく知識がない場合は、隅々まで注意してお読みください。

第 13 章「投機トレース」では、データを「投機的に」トレースバッファーにコミットする DTrace 機能について説明します。この章は、考慮中の問題との関連性がまだ明らかでない段階でデータトレースを行う、DTrace のユーザーを対象としています。

第 14 章「dtrace(1M) ユーティリティー」では、dtrace コマンド行ユーティリティーのリファレンス情報を提供します。この情報は、オンラインのマニュアルページに類似した内容です。本書で紹介するさまざまなコマンド行オプションの確認用としてご利用ください。第 15 章「スクリプトの作成」では、dtrace ユーティリティーを使って実行可能な D スクリプトを作成し、コマンド行引数を処理する方法について説明します。さらに、第 16 章「オプションとチューニング可能パラメータ」では、コマンド行または D プログラム内でチューニング可能なオプションについて説明します。

第 17 章「dtrace プロバイダ」から第 32 章「fasttrap プロバイダ」までの各章では、Solaris システムのさまざまな局面を計測する場合に利用できる DTrace「プロバイダ」について説明します。これらの章はすべてのユーザーを対象としています。これらの章を通してさまざまなプロバイダについて大まかに学んだあと、必要に応じて各章を精読してください。

第 33 章「ユーザープロセスのトレース」では、DTrace を使ってユーザープロセスを計測する例を紹介します。第 34 章「ユーザーアプリケーション向けの静的に定義されたトレース」では、アプリケーションプログラマがカスタマイズした DTrace プロバイダとプローブをユーザーアプリケーションに追加する方法について説明します。これらの章は、ユーザープログラムの開発者や管理者で、DTrace を使ってユーザープロセスの動作を検査するユーザーを対象としています。

第 35 章「セキュリティー」以降の章では、DTrace のセキュリティ、バージョン管理、安定性の属性など、上級者向けのトピックを扱います。DTrace を使ったブート時および事後トレースの実行方法については、ここで学習します。これらの章は、DTrace の上級ユーザーを対象としています。

関連情報

以下に、DTrace の使用に際して参考となる関連書籍と論文を記載します。

- 『プログラミング言語 C』、Brian W. Kernighan、Dennis M. Ritchie 共著、共立出版発行、第 2 版、1989 年、ISBN 4–320–02692–6
- Vahalia、Uresh 著、Uresh Vahalia 著、ピアソン・エデュケーション発行、2000 年、ISBN 4-89471-189-3
- Mauro、Jim and McDougall、Richard 著、『Solaris Internals: Core Kernel Components』、ピアソン・エデュケーション発行、2001 年、ISBN 4-89471-458-2

Web 上の DTrace コミュニティー (http://www.sun.com/bigadmin/content/dtrace/) では、DTrace の利用経験やスクリプトをその他のユーザーと共有できます。

マニュアル、サポート、およびトレーニング

Sun の Web サイトでは、次のサービスに関する情報も提供しています。

- マニュアル (http://jp.sun.com/documentation/)
- サポート (http://jp.sun.com/support/)
- トレーニング (http://jp.sun.com/training/)

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

コード例は次のように表示されます。

- C シェル

```
machine_name% command y|n [filename]
```

- C シェルのスーパーユーザー

```
machine_name# command y|n [filename]
```

- Bourne シェルおよび Korn シェル

```
$ command y|n [filename]
```

- Bourne シェルおよび Korn シェルのスーパーユーザー

```
# command y|n [filename]
```

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

◆ ◆ ◆ 第 1 章

1 はじめに

この章では、Solaris オペレーティングシステムの動的トレース機能について説明します。DTrace は、システムの動作を把握したいと考えているユーザーにとって理想的なツールです。Solaris に組み込まれた包括的な動的トレース機能、DTrace を利用して、管理者や開発者は、実行中の本稼働システム上で、ユーザープログラムやオペレーティングシステムの動作を検査できます。DTrace には、システムの動作を調べる機能、複数のソフトウェア層でパフォーマンスの問題を検出する機能、システムの異常な動作の原因を突き止める機能などがあります。これから見ていくように、DTrace では、システムを動的に計測し、DTrace D プログラミング言語で記述して任意の問題に対して簡潔な答えを迅速に導き出すような、独自のカスタムプログラムを作成することもできます。この章の最初の部分では、DTrace の概要と、単純な D プログラムの作成方法について説明します。残りの部分では、D プログラミングの規則の全容と、システムを詳しく分析する手順およびヒントを紹介します。Web 上の DTrace コミュニティー (http://www.sun.com/bigadmin/content/dtrace/) では、DTrace の利用経験やスクリプトをその他のユーザーと共有できます。このマニュアルで紹介するすべてのスクリプト例は、Solaris システムの `/usr/demo/dtrace` ディレクトリに収められています。

入門ガイド

DTrace では、オペレーティングシステムカーネルとユーザープロセスに動的に変更を加え、「プローブ」と呼ばれる特定の場所に指定の追加データが記録されるようにすることで、ソフトウェアシステムの情報を得ることができます。プローブとは、DTrace が一連の「アクション」(スタックトレースの記録、タイムスタンプの記録、関数の引数の記録など) の実行要求を結合する場所またはアクティビティを指します。プローブは、Solaris システム内の要所要所に配置された、プログラミング可能な「センサー」として機能します。何が起こっているか確認したい場合は、DTrace を使って、該当する情報が適切なセンサーに記録されるようにプログラミングします。この結果、DTrace は、プローブが「起動」するたびにプローブからデータを収集し、ユーザーに報告します。プローブにアクションを指定しないと、DTrace はプローブの起動を記録するだけです。

DTrace のプローブには、2 つの名前があります。1 つは一意の整数値の ID、もう 1 つは人間が読める形式の文字列名です。以下では、手始めに、新しいトレース要求を発行するたびに 1 回だけ起動するプローブ BEGIN を使って、ごく単純な要求を作成します。文字列名を指定してプローブを有効にするには、dtrace(1M) ユーティリティーの -n オプションを使用します。次のコマンドを入力します。

```
# dtrace -n BEGIN
```

しばらくすると、1 個のプローブが有効になったというメッセージに続いて、BEGIN プローブの起動を知らせる行が出力されます。この出力のあと、dtrace は、ほかのプローブの起動を待機して一時停止した状態になります。ここでは、ほかに有効なプローブがなく、BEGIN は 1 回しか起動しないため、Control-C キーを押して dtrace を終了し、シェルプロンプトに戻ってください。

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

この出力から BEGIN という名前のプローブが 1 回起動したことがわかり、その名前と整数値の ID (ここでは 1) が表示されています。デフォルトでは、このプローブが起動した CPU の整数名も出力されます。この例の CPU の欄を見ると、dtrace コマンドがプローブの起動時に CPU 0 で実行されていたことがわかります。

DTrace 要求の作成時には、任意の数のプローブおよびアクションを使用できます。以下では、上記のコマンド例に END プローブを追加して、2 つのプローブを使った単純な要求を作成してみましょう。END プローブは、トレースの完了時に 1 回だけ起動します。次のコマンドを入力し、BEGIN プローブの出力行が表示されたら、先ほどと同様に Control-C キーを押します。

```
# 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
#
```

Control-C キーを押すと、dtrace が終了し、END プローブが引き起こされます。dtrace は、END プローブの起動を報告してから終了します。

プローブを指定し、有効にする方法がわかったところで、今度は、おなじみの「Hello, World」プログラムの DTrace 版を作成してみましょう。DTrace 実行文は、コマンド行に入力するだけでなく、D プログラミング言語を使ってテキストファイ

ルに記述することもできます。ここでは、テキストエディタで hello.d という名前の新しいファイルを作成し、以下の D プログラムを記述します。

例 1–1 hello.d: D プログラミング言語で記述した “Hello, World” プログラム

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

プログラムを保存し、dtrace -s オプションを使って実行します。次のコマンドを入力します。

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

先ほどの例と同様の出力に続いて、“hello, world” というテキストが出力されます。先ほどの例とは異なり、ここでは、しばらく待つ必要も Control-C キーを押す必要もありません。これは、*hello.d* ファイル内の BEGIN プローブに、「アクション」が指定されたからです。では、D プログラムの構造をもう少し掘り下げて、何が起こっているのか確認していきましょう。

D プログラムは、有効化の対象となるプローブ (複数可) について記述する複数の「節」と、プローブの起動時に実行するアクションの集合 (オプション) で構成されます。アクションは、プローブ名に続く中括弧 ({ }) の中に、連続する文として列挙されます。各文の末尾には、セミコロン (;) が付きます。最初の文では、関数 trace() を使って、BEGIN プローブの起動時に指定の引数 (ここでは文字列「hello, world」を記録し、出力するように、DTrace に指示しています。2 番目の文では、関数 exit() を使って、トレースを中断し、dtrace コマンドを終了するように指示しています。DTrace は、trace() や exit() のように、D プログラム内で呼び出せる便利な関数を提供しています。関数を呼び出すには、関数名に続いて、丸括弧内に引数を指定します。すべての D 関数については、第 10 章「アクションとサブルーチン」を参照してください。

C プログラミング言語の知識をお持ちのユーザーは、ここまでの例や名前から、DTrace の D プログラム言語が C に似ていることに気が付いていることでしょう。実は、D 言語は、C 言語のサブセットにトレース用の特殊な関数と変数を組み合わせた言語なのです。D 言語の機能の詳細については、第 2 章以降で学習します。以前に C プログラミングをしたことがあるなら、D 言語でトレースプログラムを作成するために、その知識をほぼそのまま流用できます。もちろん、C プログラミングの経験がなくても、D プログラミングは決して難しいものではありません。D プログラミングの全構文は、この章で学ぶことができます。言語の規則について学ぶ前に、ま

ずDTraceの機能についてもう少し詳しく見ていきましょう。そのあとで上記より複雑なDプログラムの作成方法を学んでいきます。

プロバイダとプローブ

上記の例では、BEGINとENDという2つの単純なプローブの使用方法について学習しました。これらのプローブは、どこから提供されたものなのでしょうか。DTraceのプローブは、「プロバイダ」と呼ばれるカーネルモジュールのセットから提供されています。プロバイダはそれぞれ、プローブを作成して特定の計測機能を実行します。各プロバイダは、DTraceの使用時に、DTraceフレームワークに提供可能なプローブを発行するチャンスを与えられます。その後、ユーザーは、トレースアクションを有効にし、発行された任意のプローブに結合します。システム上で使用可能な全プローブを一覧するには、次のコマンドを入力します。

```
# 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 ...

#
```

すべての内容が出力されるまで、多少時間がかります。プローブの総数を表示するには、次のコマンドを入力します。

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

プローブの数は、使用するオペレーティングプラットフォームと、インストールされているソフトウェアの種類によって異なります。このため、このコマンドで出力される値は、マシンによって異なります。利用可能なプローブは膨大な数にのぼります。これらを利用して、システムの動作を隅々まで詳しく確認できます。厳密には、上のコマンドで出力された数に加えて、その他のプローブも利用できます。あとで見ていきますが、一部のプロバイダは、ユーザーのトレース要求に応じて直接、新しいプローブを作成できます。したがって、利用可能なDTraceプローブは、事実上無限にあることになります。

端末ウィンドウ内の**dtrace -l**の出力をもう1度見てください。前述したように、各プローブには、整数値のIDと人間が読める形式の名前があります。人間が読める形

式の名前は 4 つの部分で構成され、dtrace の出力中で個別の列に表示されます。プローブ名を構成する 4 つの部分は、次のとおりです。

プロバイダ	このプローブを発行した DTrace プロバイダの名前。プロバイダ名は、通常、プローブを有効にする計測機能を実行する DTrace カーネルモジュールの名前と一致しています。
モジュール	プローブが置かれているモジュールの名前 (このプローブが特定のプログラムの場所である場合)。この名前は、カーネルモジュール名またはユーザーライブラリ名です。
機能	プローブが置かれているプログラム関数の名前 (このプローブが特定のプログラムの場所である場合)。
名前	BEGIN や END のように、プローブの機能を連想しやすい名前。プローブ名の最後の部分です。

人間が読める形式のプローブ名を完全な形で記述するときは、4 つの部分をコロンで区切ります。

provider:*module*:*function*:*name*

一覧に表示されるプローブの中には、先ほど使用した BEGIN や END のように、モジュールと関数を持たないものもあります。これらのプローブは、特定の計測機能付きプログラムの関数や場所を表すものではないので、モジュールのフィールドと関数のフィールドが空欄になっています。これらのプローブは、「トレース要求の終了」など、より抽象度の高い概念を表しています。名前の構成要素としてモジュール名と関数名を持つプローブを「アンカーされたプローブ」、これらの名前を持たないプローブを「アンカーされていないプローブ」と呼びます。

要求時にプローブ名の一部を省略して指定した場合、DTrace は通常、指定された名前と一致する値を持つすべてのプローブを検出します。たとえば、BEGIN と指定した場合、DTrace は、プロバイダフィールド、モジュールフィールド、関数フィールドの値とは関係なく、BEGIN という名前フィールドを持つプローブをすべて検出します。上記の例では、たまたま一致する文字列を持つプローブが 1 個しかなかったため、同じ結果になりました。BEGIN プローブの正式な名前は dtrace:::BEGIN です。この名前が示すように、このプローブは DTrace フレームワーク自体から提供され、関数にアンカーされていません。したがって、hello.d プログラムは、次のように書き換えることができます。結果は、上記の例と同じです。

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

これで、プローブの提供元と、プローブの指定方法がわかりました。以下では、プローブを有効にし、DTrace に何らかの処理を指示したときの様子について少し詳しく見ていきます。その後、D プログラミング言語の解説に戻ります。

コンパイルと計測機能

Solaris で従来のプログラムを作成するときは、プログラムを記述したあと、コンパイラを使って、ソースコードを実行可能なオブジェクトコードに変換します。`dtrace` コマンドの使用時には、先ほど `hello.d` プログラムの記述に使用した D 言語用のコンパイラが呼び出されます。コンパイルの完了後、プログラムは DTrace で実行するためにオペレーティングシステムカーネルに送信されます。ここで、プログラムに指定されたプローブが有効になり、対応するプロバイダにより、これらのプローブをアクティブにするために必要な計測機能が実行されます。

DTrace の計測機能はすべて、完全に動的です。 使用されるプローブだけが、個別に有効になります。アクティブでないプローブに、計測機能用コードはありません。このため、DTrace を使用していないときに、システムのパフォーマンスが低下することはありません。計測が完了し、`dtrace` コマンドが終了すると、それまで使用していたすべてのプローブが自動的に無効になり、その計測機能が削除されます。こうして、システムは完全に元の状態に戻ります。DTrace がアクティブになっていないときのシステムの状態は、DTrace ソフトウェアがインストールされていないときのシステムの状態と実質的に同じです。

各プローブの計測機能は、稼働中のオペレーティングシステムまたはユーザーが選択したユーザープロセスに対して動的に実行されます。システムは休止または停止することなく、計測機能コードは有効になったプローブにのみ追加されます。このため、DTrace を使用したプローブの影響は、ユーザーが DTrace に指示した内容だけに制限されます。 関係のないデータがトレースされたり、システム内で一元的な「トレーススイッチ」がオンになったりすることはありません。このように、すべての DTrace 計測機能は、できるだけ能率よく機能するように設計されています。DTrace のこれらの機能を利用すれば、システムを停止することなく、リアルタイムで問題を解決できます。

DTrace フレームワークは、任意の数の仮想クライアントをサポートします。DTrace 計測や DTrace コマンドは、システムのメモリー容量が許すかぎり、好きな数だけ同時に実行できます。なお、DTrace コマンドは、配下にある共通の計測機能を使用しつつ、それぞれ独立して処理を行います。このため、1 つのシステム上で同時に DTrace を利用できるユーザーの数は、事実上無制限になります。 開発者、管理者、サービス担当者は、共同で作業できるだけでなく、同一システム上のさまざまな問題に、互いに干渉することなく個別に対処できます。

DTrace D プログラムは、安全な中間形式にコンパイルされたあと、プローブの起動時に実行されます。この点で、C や C++ より、Java™ プログラミング言語で書かれたプログラムによく似ています。この中間形式の安全性は、DTrace カーネルソフト

ウェアがこのプログラムを最初に検査するときに検証されます。また、DTrace 実行環境は、D プログラムの実行中に発生する実行時エラー (ゼロ除算、無効なメモリーの間接参照など) を処理し、その結果をユーザーに報告します。このため、信頼性の低いプログラムを作成して、Solaris カーネルやシステムで実行中のプロセスに DTrace が害を及ぼす危険を回避できます。これらの安全性機能により、本稼働環境でシステムのクラッシュや破損を心配することなく DTrace を使用できます。プログラミングミスを犯しても、DTrace がエラーを報告し、計測機能を無効にするため、ユーザーはエラー箇所を修正し、再試行するだけで済みます。DTrace のエラー報告機能とデバッグ機能については、あとで説明します。

以下の図に、プロバイダ、プローブ、DTrace カーネルソフトウェア、`dtrace` コマンドをはじめとする、DTrace アーキテクチャの各種構成要素を示します。

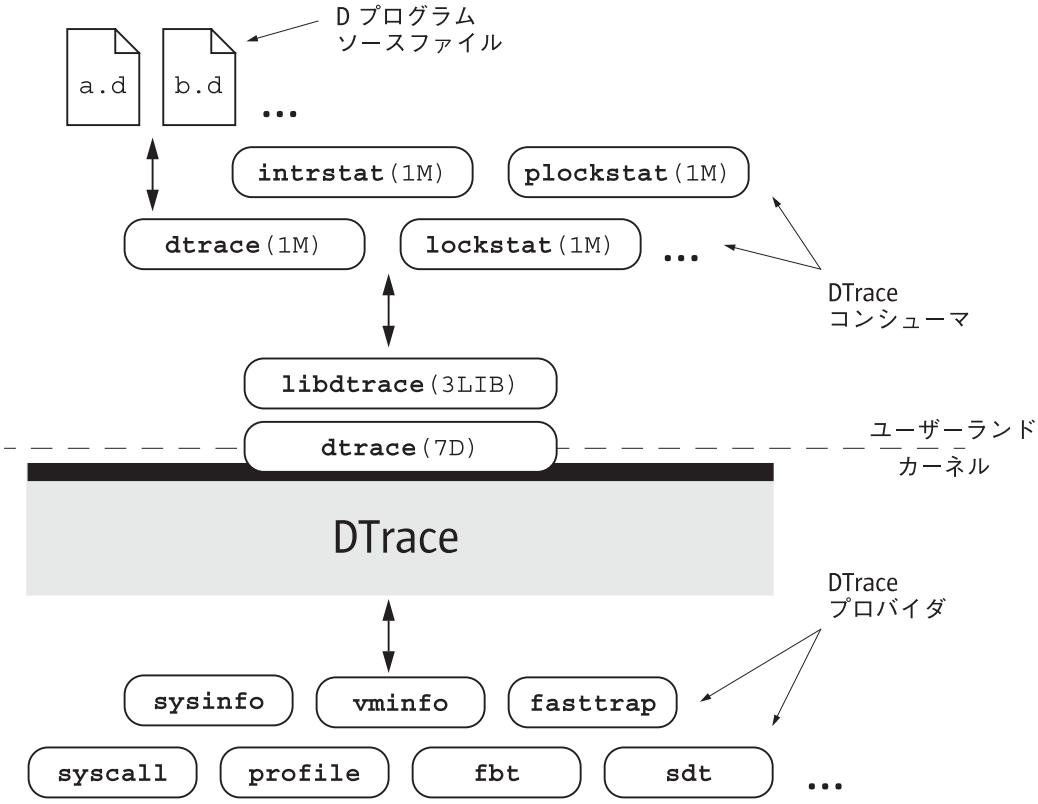

図 1–1　DTrace のアーキテクチャと構成要素の概要

DTrace の機能について理解できたところで、D プログラミング言語の解説に戻り、いくつかのプログラムを作成してみましょう。

変数と算術式

次のプログラム例では、DTrace の `profile` プロバイダを使って、単純な時間ベースのカウンタを実装します。profile プロバイダは、D プログラム内の記述に基づいて新しいプローブを作成できます。たとえば、`profile:::tick-`*n*`sec` (*n* は整数) という名前のプローブを作成した場合、profile プロバイダは、*n* 秒ごとに起動するプローブを 1 つ作成します。次のソースコードを記述し、`counter.d` という名前のファイルに保存します。

```
/*
 * 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);
}
```

実行すると、Control-C キーを押すまでの経過秒数がカウントされ、最後に合計が出力されます。

```
# 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
#
```

プログラムの最初の 3 行は、プログラムの内容を説明するコメント行です。C、C++、および Java プログラミング言語の場合と同様に、D コンパイラは `/*` と `*/` で囲まれた文字列をすべて無視します。コメントは、D プログラム内のどこでも使用できます。プローブ節の中でも外でも使用できます。

BEGIN プローブ節では、次の文により、i という名前の新しい変数が定義され、整数値 0 が割り当てられています。

```
i = 0;
```

C、C++、および Java プログラミング言語の場合とは異なり、D の変数は、プログラム文の中で使用するだけで作成されます。明示的変数宣言は不要です。変数の型は、この変数をプログラム内ではじめて使用したとき、そこに代入された値の型になります。各変数は 1 つの型を持ち、プログラムの完了時まで変わりません。したがって、2 回目以降の使用時にも、最初に代入された値と同じ型の値を参照させる必要があります。たとえば、counter.d の変数 i に最初に代入される値は、整数定数 0 です。そのため、この変数の型は int になります。D は、C と同じ基本的な整数データ型を提供します。たとえば、次のデータ型があります。

char	文字型または単一バイトの整数型
int	デフォルト整数型
short	短整数型
long	長整数型
long long	拡張された長整数型

これらのデータ型のサイズは、オペレーティングシステムカーネルのデータモデル (第 2 章「型、演算子、および式」を参照) によって異なります。また D では、オペレーティングシステムごとに定義されている各種データ型と同様に、さまざまな固定長の符号付き/符号なし整数型に、覚えやすい組み込み名が付けられています。

counter.d の中央部分には、カウンタ i の値を増分するプローブ節があります。

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

この節には、使用可能なプロセッサ上で毎秒 1 回起動する新しいプローブを作成するように profile プロバイダに指示する、profile:::tick-1sec という名前のプローブが指定されています。この節に含まれる 2 つの文のうち、最初の文は以前の値に 1 を加算して i に代入し、2 番目の文は新しい i の値をトレースします。D では、C の一般的な算術演算子をすべて使用できます。使用可能な算術円算子のリストは、第 2 章「型、演算子、および式」に記載されています。また C の場合と同様に、++ 演算子は、対応する変数を 1 増分したい場合に使用します。trace() 関数は、引数として任意の D 式を取ります。この考えに従って counter.d をさらに簡潔に書き直すと、次のようになります。

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

変数 i の型を明示的に制御したい場合は、型名を丸括弧で囲んで割り当てます。こうすれば、整数 0 がその型に「キャスト」されます。たとえば、D で char の最大サイズを決定したい場合は、BEGIN 節を次のように変更します。

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

counter.d の実行を開始してしばらくすると、トレース値が大きくなり、その後再びゼロに戻ります。値がゼロに戻るのを待てない場合は、profile プローブ名を profile:::tick-100msec に変更してみてください。こうすれば、カウンタの値は 100 ミリ秒に 1 回 (毎秒 10 回) のペースで増分するようになります。

述語

D とその他のプログラミング言語 (C、C++、Java など) のもっとも顕著な違いは、D では、if 文やループに代表される制御フロー構文が使用されない点にあります。D プログラム節は、決まった量のデータ (オプション) をトレースする単一の文のリストとして記述されます。D では、プログラム節の前に置かれる「述語」という論理式を使って、条件付きでデータをトレースしたり、制御フローを変更したりできます。述語式は、プローブの起動時、対応する節に関連付けられた文が実行される前に評価されます。述語の評価の結果が真 (ゼロ以外の値) なら、文のリストが実行されます。述語の評価の結果が偽 (ゼロ) なら、文は実行されず、プローブの起動は無視されます。

次のソースコードを入力し、countdown.d という名前のファイルに保存してください。

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

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

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

このDプログラムは、述語を使用することにより、10秒間のカウントダウンを行うタイマーを実装しています。countdown.dを実行すると、10からカウントダウンが開始され、メッセージが表示されたあと、プログラムは終了します。

```
# 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!
#
```

このプログラムは、まずBEGINプローブを使って、整数iをカウントダウンの開始値10に初期化します。次に、前記の例と同じくtick-1secプローブを使って、毎秒1回起動するタイマーを実装します。countdown.dでは、tick-1secプローブが述語とアクションリストの異なる2つの節に記述されている点に注目してください。述語は、プローブ名と節の文が入っている中括弧({ })との間に置かれた論理式です。前後をスラッシュ(/ /)で囲まれた形式になっています。

最初の述語は、iがゼロより大きいかどうか(タイマーがまだ実行中かどうか)をテストします。

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

関係演算子>は、「-より大きい」を意味し、偽の場合は整数値0、真の場合は1を返します。Dでは、Cの関係演算子がすべてサポートされます。使用可能な関係演算子のリストは、第2章「型、演算子、および式」に記載されています。iがまだゼロになっていない場合、スクリプトはiをトレースし、--演算子を使って、値を1減らします。

2 番目の述語は、== 演算子を使って、i が 0 (カウントダウンが完了した) の場合に真を返します。

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

最初の hello.d の例の場合と同様に、 countdown.d は、二重引用符で囲まれた文字列 (「文字列定数」) を使って、カウントダウン完了時のメッセージを出力します。次に、exit() 関数を使って dtrace を終了し、シェルプロンプトに戻ります。

countdown.d の構造をよく見ると、プローブの記述は同じであるが述語とアクションが異なる 2 つの節を作成することによって、次のような論理フローが作成されていることがわかります。

```
i = 10
毎秒 1 回
  i がゼロより大きい場合
     trace(i--);
  i がゼロの場合
     trace("blastoff!");
     exit(0);
```

述語を使って複雑なプログラムを作成するときは、まずこの方法でアルゴリズムを視覚化した上で、それぞれの条件構文を別々の節および述語に変換してみることをお勧めします。

では次に、述語を新しいプロバイダ syscall と組み合わせて、実際に D トレースプログラムを作成してみましょう。syscall プロバイダには、任意の Solaris システムコールの開始時または終了時にプローブを有効にする機能があります。次の例では、DTrace を使って、シェルがシステムコール read(2) または write(2) を実行する様子を観察します。まず、端末ウィンドウを 2 つ開きます。1 つは DTrace 用、もう 1 つは監視するシェルプロセス用です。後者のウィンドウに次のコマンドを入力して、このシェルのプロセス ID を取得します。

```
# echo $$
12345
```

1 つ目の端末ウィンドウに戻り、次の D プログラムを入力して、rw.d という名前のファイルに保存します。プログラムを入力する際、*12345* は、シェルのプロセス ID (echo コマンドの実行結果) で置き換えてください。

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

```
{

}
```

rw.d のプローブ節の本体には、何も入力しません。これは、このプログラムではプローブの起動通知をトレースするだけで、ほかには何もトレースしないからです。rw.d の内容を入力できたら、dtrace を使って計測を開始します。その後、2 つ目のウィンドウでいくつかのコマンドを実行します。なお、コマンドを 1 つ入力したら、そのたびに Return キーを押す必要があります。コマンドの実行に合わせて、1 つ目のウィンドウに、dtrace からのプローブ起動通知が表示されます。以下の出力例を参照してください。

```
# 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
...
```

この出力から、シェルがシステムコール read(2) と write(2) を実行することにより、端末ウィンドウから文字を読み取り、結果をエコーバックしていることがわかります。この例では、これまでに説明した概念のほかに、いくつかの新しい概念が含まれています。まず、read(2) と write(2) を同じ方法で計測するため、複数のプローブ記述をコンマで区切った形式の単一のプローブ節が使用されています。

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

読みやすさを考慮して、これらのプローブ記述は 2 行に分けて入力されています。この配置は必須ではありませんが、より読みやすいスクリプトを作成するのに役立ちます。次に、該当のシェルプロセスが実行するシステムコールだけを検出する述語が定義されています。

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

この述語では、事前定義済みの DTrace 変数 pid (対応するプローブを起動したスレッドのプロセス ID が入る) が使用されています。DTrace には、プロセス ID のような使用頻度の高い情報を導き出す変数定義が多数組み込まれています。以下に、基本的な D プログラムの作成に役立つ DTrace 変数をいくつか紹介しておきます。

変数名	データ型	意味
errno	int	システムコールの現在の errno 値
execname	string	現在のプロセスの実行可能ファイルの名前
pid	pid_t	現在のプロセスのプロセス ID
tid	id_t	現在のスレッドのスレッド ID
probeprov	string	現在のプローブ記述のプロバイダフィールド
probemod	string	現在のプローブ記述のモジュールフィールド
probefunc	string	現在のプローブ記述の関数フィールド
probename	string	現在のプローブ記述の名前フィールド

作成した計測機能プログラムを使用し、プロセス ID と計測するシステムコールプローブを変更して、システム上で実行中のさまざまなプロセスを計測してみてください。その後、もう少し簡単な変更を加えることで、rw.d を truss(1) のような非常に単純なシステムコールトレースツールに変更できます。何も記述されていないプローブ記述フィールドは、「あらゆるプローブ」を表すワイルドカードとして機能します。上記のプログラムを次のような新しいソースコードに変更することによって、シェルで実行されるあらゆるシステムコールをトレースできるようになります。

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

}
```

シェルに cd、ls、date などのコマンドを入力して、DTrace プログラムからの報告内容を確認してください。

出力書式

システムコールをトレースすることにより、ほとんどのユーザープロセスの動作を効果的に監視できます。管理者または開発者として Solaris の truss(1) ユーティリティーを使用したことがあるユーザーなら、このユーティリティーが問題の解決に役立つ手軽で便利なツールであることをすでにご存知でしょう。truss をまだ使ったことがないユーザーも、シェルに次のコマンドを入力して、今すぐその効果を試してください。

```
$ truss date
```

date(1) によって実行されるすべてのシステムコールのトレース結果が書式付きで表示され、最後の行に date(1) そのものの出力が表示されます。次の例は、上記の rw.d プログラムに truss(1) とよく似た出力書式を設定し、出力を読みやすくしたものです。以下のプログラムを入力し、trussrw.d という名前のファイルに保存してください。

例 1–2 trussrw.d: truss(1) の出力書式を設定したシステムコールトレースプログラム

```
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);
}
```

この例では、すべての述語の定数 12345 の部分が、ラベル $1 で置き換えられています。このラベルを使用することで、監視対象のプロセスをスクリプトの引数として指定できます。 $1 には、スクリプトのコンパイル時に、最初の引数の値が代入されます。trussrw.d を実行するには、dtrace の -q オプションと -s オプションを指定したあと、最後の引数としてシェルのプロセス ID を指定します。-q オプションを指定すると、dtrace が非出力モードになり、これまでの例で出力されていた見出し行と CPU 列および ID 列が出力されなくなります。つまり、出力されるのは、明示的にトレースしたデータだけになります。指定のシェルウィンドウで次のコマンドを入力し (その際、12345 の部分はシェルプロセスのプロセス ID で置き換える)、指定したシェルで、Return キーを数回押します。

```
# 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
#
```

ここからは、D プログラムとその出力についてさらに詳しく見ていきます。まず、上記のプログラムとよく似た節により、シェルで呼び出されている read(2) と write(2) が計測されます。ただし、この例では、データのトレースと結果の書式付き出力に、新しい関数 printf() が使用されています。

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

printf() 関数は、以前に使用した trace() 関数のようにデータをトレースする機能と、データとその他のテキストを指定された書式で出力する機能を兼ね備えています。DTrace は、printf() 関数からの指示を受けて、2 番目以降の各引数に関連付けられたデータをトレースし、最初の printf() 引数 (「書式設定文字列」) で指定された規則に従って結果を書式設定します。

書式設定文字列は、% で始まる任意の数の書式変換を含む標準文字列で、対応する引数の書式を指定します。この例の書式設定文字列の最初の変換は 2 番目の printf() 引数、2 番目の変換は 3 番目の引数 (以降同様) に対応しています。変換と変換の間のすべてのテキストは、変更されずそのまま出力されます。% 変換文字に続く文字は、対応する引数の書式を表します。以下に、trussrw.d で使用されている 3 つの書式変換の意味を示します。

%d	対応する値を 10 進整数として出力
%s	対応する値を文字列として出力
%x	対応する値を 16 進整数として出力

DTrace の printf() の機能は、C の printf(3C) ライブラリルーチンやシェルの printf(1) ユーティリティーと同じです。以前に printf() を使ったことがない場合は、第 12 章「出力書式」にその書式とオプションの詳細が記載されているので確認してください。この章は、すでに別の言語の printf() を使用したことがあるユーザーも、注意深く読む必要があります。D の printf() は組み込み関数であり、DTrace 専用の新しい書式変換を利用できます。

D コンパイラには、1 つ 1 つの printf() 書式設定文字列をその引数リストと照合することにより、プログラミングミスを防ぐ機能があります。上記の例の節にある

probefunc を整数 123 に変更してみてください。変更後のプログラムを実行すると、文字列書式変換 %s では整数引数を扱うことができないことを示すエラーメッセージが表示されます。

```
# 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
#
```

システムコール read または write の名前とその引数を出力するには、次のような printf() 文を使用します。

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

この文は、現在のプローブ関数の名前とシステムコールの最初の 3 つの整数引数 (DTrace 変数 arg0、arg1、および arg2) をトレースします。プローブ引数の詳細は、第 3 章「変数」を参照してください。read(2) と write(2) の最初の引数はファイル記述子を表しており、10 進数として出力されます。2 番目の引数はバッファーアドレスを表しており、16 進数値として書式設定されています。最後の引数はバッファーサイズを表しており、10 進数値として書式設定されています。3 番目の引数で使用されている書式指定子 %4d は、%d 書式変換を使用し、最小フィールド幅を 4 文字に制限して値を出力するように指定しています。4 文字未満の整数であれば、printf() によって適切な数の空白文字が挿入され、出力の配置が調整されます。

システムコールの結果を出力し、各出力行を完成させるには、次の節を使用します。

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

また、syscall プロバイダは、システムコールごとに、entry プローブと return プローブの両方を発行します。syscall プロバイダの return プローブの DTrace 変数 arg1 には、システムコールの戻り値が入ります。この戻り値は、10 進整数として書式設定されます。書式設定文字列内のバックスラッシュで始まる文字シーケンスは、タブ (\t) と改行 (\n) を表しています。これらの「エスケープシーケンス」は、入力するのが難しい文字を出力または記録する場合に使用します。D では、C、C++、および Java プログラミング言語と共通のエスケープシーケンスがサポートされています。すべてのエスケープシーケンスについては、第 2 章「型、演算子、および式」を参照してください。

配列

D では、整数型の変数のほか、文字列型や複合型を表す型として、「構造体」と「共用体」を定義できます。D では、C で使用できるすべての型を使用できます。C プログラミングの経験がないユーザーは、第 2 章「型、演算子、および式」で、さまざまなデータ型について確認してください。D では、「連想配列」と呼ばれる特殊な変数もサポートされています。連想配列は、キーの集合を値の集合に関連付けるという点では通常の配列と同じですが、キーが固定範囲の整数値に制限されないという特徴を備えています。

D の連想配列には、1 個以上の任意の型の値からなるリストでインデックスを付けることができます。個々のキー値が集まって「組」を形成し、この組が、配列へのインデックス付けと、個々のキーに対応する値の参照または変更に使用されます。連想配列で使用される組はすべて、同じ型署名を使用している必要があります。これは、すべての組キーの長さ、キー型、およびキーの並び順が一致していなければならないということです。連想配列の各要素に関連付けられる値も、配列全体の固定型になります。たとえば、次の D 文は、組署名が `[ string, int ]`、値の型が `int` の新しい連想配列 `a` を定義し、この配列に整数値 456 を格納します。

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

配列の定義後は、その他の D 変数と同じ方法で、配列要素にアクセスできます。たとえば、次の D 文は、値を 456 から 457 に増分することにより、以前に `a` に格納された配列要素を変更します。

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

配列要素の値がまだ割り当てられていない場合、値はゼロに設定されます。では、実際に D プログラム内で連想配列を使用してみましょう。以下のプログラムを入力し、`rwtime.d` という名前のファイルに保存してください。

例 1–3　`rwtime.d`: `read(2)` および `write(2)` 呼び出しの時間

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
    ts[probefunc] = timestamp;
}

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

rwtime.d の実行時には、trussrw.d のときと同じように、シェルプロセスの ID を指定します。シェルコマンドをいくつか入力すると、各システムコールの経過時間が表示されます。次のコマンドを入力し、別のシェルで Return キーを何回か押します。

```
# 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
#
```

各システムコールの経過時間をトレースするには、read(2) と write(2) の entry (開始時) および return (終了時) を両方とも計測し、各ポイントで時間を確認する必要があります。次に、システムコールの終了時に、最初のタイムスタンプと 2 番目のタイムスタンプの差分を計算します。システムコールごとに別々の変数を使用することもできますが、そうすると新たにシステムコールを追加してプログラムを拡張するのが難しくなります。そのため、プローブ関数名でインデックスが付けられた連想配列を使用します。以下は、最初のプローブ節です。

```
syscall::read:entry,
syscall::write:entry
/pid == $1/
{
    ts[probefunc] = timestamp;
}
```

この節では、ts という名前の配列が定義され、適切なメンバーに DTrace 変数 timestamp の値が割り当てられます。この変数は、Solaris ライブラリルーチン gethrtime(3C) とよく似た機能を持っており、常に増加するナノ秒カウンタの値を返します。entry のタイムスタンプが保存されたあと、対応する return プローブにより、再度 timestamp が調べられ、現在の時間と保存された値の差分が報告されます。

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

return プローブの述語は、DTrace がトレースしているプロセスが適切であることと、対応する entry プローブがすでに起動していて、ts[probefunc] にゼロ以外の値が割り当てられていることを要求します。この方法で、DTrace がはじめて起動したときの無効な出力を排除できます。dtrace の実行時に、すでにシェルが read(2) システムコール内で入力を待っている場合、最初の read(2) の read:entry を飛ばして、read:return プローブが起動します。このとき、まだ割り当てられていない ts[probefunc] の値はゼロになります。

外部のシンボルと型

DTrace 計測機能は、Solaris オペレーティングシステムカーネルの内部で実行されます。このため、特殊な DTrace 変数やプローブ引数だけでなく、カーネルデータの構造、シンボル、および型にもアクセスできます。DTrace の上級ユーザー、管理者、サービス担当者、ドライバ開発者は、この機能を利用して、オペレーティングシステムカーネルやデバイスドライバの低レベルの動作を検査できます。Solaris オペレーティングシステムの内部構造については、このマニュアルの冒頭で紹介した関連書籍を参照してください。

D では、オペレーティングシステムには定義されているが、D プログラム内では定義されていないシンボルにアクセスする際、特別なスコープ演算子として逆引用符 (‘) を使用します。たとえば、Solaris カーネルには、メモリーアロケータのデバッグ機能を有効にする、チューニング可能なシステム変数 kmem_flags の C 宣言が含まれています。kmem_flags の詳細は、『Solaris カーネルのチューンアップ・リファレンスマニュアル』を参照してください。このチューニング可能な変数は、次のように、カーネルソースコード内に C で宣言されています。

```
int kmem_flags;
```

この変数の値を D プログラム内でトレースする場合は、次のような D 文を記述します。

```
trace(‘kmem_flags);
```

DTrace は、カーネルシンボルに、対応するオペレーティングシステム C コードで使用されている型を関連付けます。このため、ネイティブのオペレーティングシステムデータ構造に、ソースから簡単にアクセスできます。カーネルシンボル名は、D 変数および関数識別子とは別の名前空間に格納されています。したがって、カーネルシンボル名と D 変数名が競合することはありません。

この章では、DTrace の概要を把握し、より規模が大きく複雑な D プログラムを作成するために必要な DTrace の基礎知識を身に付けました。以降の章では、D の規則の全容を明らかにし、実際に DTrace を使って、複雑なパフォーマンス測定やシステム機能分析を単純化する方法を学んでいきます。さらに、DTrace を使ってユーザーアプリケーションの動作とシステムの動作を関連付け、ソフトウェアスタック全体を分析する方法についても学びます。

DTrace の学習は、まだ始まったばかりです。

◆ ◆ ◆ 第 2 章

2 型、演算子、および式

D では、さまざまなデータオブジェクトにアクセスし、操作できます。 具体的には、変数やデータ構造を作成または変更したり、オペレーティングシステムカーネルやユーザープロセスに定義されているデータオブジェクトにアクセスしたり、整数定数、浮動小数点定数、文字列定数を宣言したりすることができます。D は、オブジェクトの操作や複合式の作成に使用される ANSI-C 演算子の上位集合を提供します。この章では、型、演算子、および式の規則の詳細を説明します。

識別子名とキーワード

D 識別子名は、アルファベットの大文字と小文字 、数字、および下線で構成されます。先頭文字は必ずアルファベットまたは下線になります。先頭文字が下線 (_) の識別子名はすべて、D システムライブラリ用に予約されています。これらの名前を D プログラム内で使用してはなりません。D プログラムでは、通常、変数名はアルファベットの大文字と小文字、定数名はすべて大文字で記述されます。

D 言語のキーワードは、プログラミング言語構文そのもので使用するために予約されている特別な識別子です。これらの名前は常にアルファベットの小文字のみで指定され、D 変数名に使用することは禁じられています。

表 2–1 D のキーワード

auto*	goto*	sizeof
break*	if*	static*
case*	import*+	string+
char	inline	stringof+
const	int	struct

表 2–1 D のキーワード (続き)

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+

D では、ANSI-C キーワードの上位集合がキーワードとして予約されています。将来 D 言語で使用するために予約されているキーワードには、アスタリスク (*) が付けられています。将来のために予約されているキーワードを使用しようとすると、D コンパイラから構文エラーが返されます。D では定義されているが ANSI-C では定義されていないキーワードには、プラス記号 (+) が付けられています。D では、ANSI-C のすべての型および演算子を使用できます。D プログラミングの主要な特徴は、制御フロー構文を使用しない点です。ANSI-C で制御フローに関連付けられたキーワードは、D では将来のために予約されています。

データ型とサイズ

D は、整数と浮動小数点定数の基本データ型を提供しています。算術演算は、D プログラム内の整数に対してのみ実行可能です。D では、データ構造を初期化するために浮動小数点定数を使用できますが、浮動小数点算術演算の使用は許可されていません。D は、プログラムの記述用として、32 ビットと 64 ビットのデータモデルを提供しています。プログラムの実行時には、稼働中のオペレーティングシステムカーネルに関連付けられたネイティブデータモデルが使用されます。システムのネイティブデータモデルは、`isainfo -b` コマンドで確認できます。

以下の表に、2 つのデータモデルの整数型名とサイズを一覧します。整数は常に、システムのネイティブバイトエンコーディングの順番で、2 の補数として表現されます。

表 2–2　D の整数データ型

型名	サイズ (32 ビット)	サイズ (64 ビット)
`char`	1 バイト	1 バイト
`short`	2 バイト	2 バイト
`int`	4 バイト	4 バイト
`long`	4 バイト	8 バイト
`long long`	8 バイト	8 バイト

整数型の前には、修飾子 `signed` または `unsigned` が付けられます。符号修飾子が存在しない場合、その型は符号付き (signed) であると見なされます。D コンパイラでは、以下のような型の別名も使用できます。

表 2–3　D 整数型の別名

型名	説明
`int8_t`	1 バイトの符号付き整数
`int16_t`	2 バイトの符号付き整数
`int32_t`	4 バイトの符号付き整数
`int64_t`	8 バイトの符号付き整数
`intptr_t`	ポインタと同じサイズの符号付き整数
`uint8_t`	1 バイトの符号なし整数
`uint16_t`	2 バイトの符号なし整数
`uint32_t`	4 バイトの符号なし整数
`uint64_t`	8 バイトの符号なし整数
`uintptr_t`	ポインタと同じサイズの符号なし整数

これらの別名は、1 つ前の表に示した基本型と対応しており、データモデルごとに定義されています。たとえば、型名 `uint8_t` は、`unsigned char` 型の別名です。D プログラム内で使用する型の別名を独自に定義する方法については、第 8 章「型と定数の定義」を参照してください。

D の浮動小数点型は、ANSI-C の宣言と型に互換性があります。D では、浮動小数点演算子はサポートされていませんが、`printf()` 関数を使って、浮動小数点型データオブジェクトのトレースおよび書式設定を行うことができます。使用可能な浮動小数点型は、次の表のとおりです。

表 2–4　D の浮動小数点データ型

型名	サイズ (32 ビット)	サイズ (64 ビット)
`float`	4 バイト	4 バイト
`double`	8 バイト	8 バイト
`long double`	16 バイト	16 バイト

D では、ASCII 文字列を表す特殊な `string` 型も使用できます。文字列については、第 6 章「文字列」で詳しく説明します。

定数

整数定数は、10 進法 (`12345`)、8 進法 (`012345`)、16 進法 (`0x12345`) で記述できます。8 進法 (base 8) の定数には、接頭辞としてゼロを付ける必要があります。16 進法 (base 16) の定数には、接頭辞として `0x` または `0X` を付ける必要があります。整数定数には、その値を表す型として、`int`、`long`、`long long` のうちもっとも小さい型が割り当てられます。負の値には、符号付きの型が割り当てられます。正の値で、符号付きの型で表すには値が大きすぎる場合は、符号なしの型が割り当てられます。D の型であることを明示的に指定するには、次のいずれかの接尾辞を割り当てます (すべての整数定数に共通)。

`u` または `U`	コンパイラによって選択された `unsigned` 型
`l` または `L`	`long`
`ul` または `UL`	`unsigned long`
`ll` または `LL`	`long long`
`ull` または `ULL`	`unsigned long long`

浮動小数点定数は常に 10 進数で記述します。さらに、小数点 (`12.345`) か指数 (`123e45`)、またはその両方 (`123.34e-5`) を含める必要があります。浮動小数点定数には、デフォルトで `double` 型が割り当てられます。D の型であることを明示的に指定するには、次のいずれかの接尾辞を割り当てます (すべての浮動小数点定数に共通)。

`f` または `F`	`float`
`l` または `L`	`long double`

文字定数は単一の文字か、エスケープシーケンスを単一引用符で囲んだ形式 ('a') で記述します。文字定数には、int 型が割り当てられます。文字定数は、ASCII 文字集合内のその文字の値に対応する値を持つ整数定数と同等です。文字とその値の対応関係は、ascii(5) のマニュアルページで確認できます。文字定数には、以下の表に示す特殊なエスケープシーケンスを含めることもできます。D では、ANSI-C と同じエスケープシーケンスがサポートされています。

表 2–5 D の文字エスケープシーケンス

\a	警告	\\	バックスラッシュ
\b	バックスペース	\?	疑問符
\f	用紙送り	\'	単一引用符
\n	改行	\"	二重引用符
\r	キャリッジリターン	\0*oo*	8 進数値 0*oo*
\t	水平タブ	\x*hh*	16 進数値 0x*hh*
\v	垂直タブ	\0	NULL 文字

複数の文字指定子を 1 組の単一引用符で囲むことにより、整数のバイトを、対応する文字指定子に合わせて個別に初期化できます。これらのバイトは、文字定数の左から順に読み取られ、使用しているオペレーティング環境のネイティブの並び順 (エンディアン) で、整数に割り当てられます。1 つの文字定数に、最大 8 個の文字指定子を含めることができます。

文字列定数は、その長さを問わず、二重引用符で囲んだ形式 ("hello") で表されます。文字列定数には、改行文字をそのまま含めることはできません。改行文字を含めたい場合は、実際に改行文字を入力する代わりに、\n というエスケープシーケンスを入力します。文字列定数には、文字定数で使用できる特殊文字エスケープシーケンス (上記参照) を含めることができます。ANSI-C の場合と同じく、文字列は、NULL 文字 (\0) で終わる文字配列で表されます。この NULL 文字は、宣言された文字列定数ごとに暗黙的に挿入されるものです。文字列定数には、D の特殊な string 型が割り当てられます。D コンパイラは、文字列として宣言された文字配列の比較とトレースを行う特別な機能セットを提供しています。この機能セットについては、第 6 章「文字列」を参照してください。

算術演算子

D プログラムでは、次の表に示す二項算術演算子を使用できます。これらの演算子で整数を操作したときの効果は、ANSI-C の場合と同じです。

表 2–6　D の二項算術演算子

+	整数の加算
-	整数の減算
*	整数の乗算
/	整数の除算
%	整数の剰余演算

D の算術演算は、整数オペランドかポインタに対してのみ実行できます (第 5 章「ポインタと配列」を参照)。D プログラムの浮動小数点オペランドに対して、算術演算を実行することはできません。DTrace 実行環境は、整数のオーバーフローまたはアンダーフローに対して、何の処理も行いません。整数のオーバーフロー、アンダーフローについては、状況に応じてユーザー自身が確認する必要があります。

DTrace 実行環境では、/ 演算子や % 演算子の誤用によるゼロ除算エラーが自動的に検査され、報告されます。D プログラムで無効な除算が実行されると、その影響を受ける計測機能が自動的に無効化され、エラーが報告されます。DTrace でエラーが検出されても、その他の DTrace ユーザーやオペレーティングシステムカーネルに悪影響が及ぶことはありません。したがって、エラーのある D プログラムを不注意で作成してしまったとしても、障害の起きる心配はありません。

上記の二項演算子のほかに、+ と - を単項演算子として使用することもできます。この 2 つの演算子は、どの二項算術演算子よりも優先されます。D の演算子の優先順位と結合規則については、表 2–11 を参照してください。優先順位を制御したい場合は、式を丸括弧 ( ) で囲んでグループ化します。

関係演算子

D プログラムでは、次の表に示す二項関係演算子を使用できます。これらの演算子の効果は、ANSI-C の場合と同じです。

表 2–7　D の関係演算子

<	左のオペランドは右のオペランドより小さい

表 2–7 D の関係演算子 (続き)

<=	左のオペランドは右のオペランド以下
>	左のオペランドは右のオペランドより大きい
>=	左のオペランドは右のオペランド以上
==	左のオペランドは右のオペランドと等しい
!=	左のオペランドは右のオペランドと等しくない

関係演算子は、D の述語の記述によく使用されます。各演算子は、int 型の値 (条件が真の場合は 1、偽の場合は 0) を返します。

関係演算子は、整数同士、ポインタ同士、または文字列同士の組に適用されます。ポインタ同士を比較した場合、双方を符号なし整数と見なして整数比較を実行した場合と同じ結果になります。文字列同士を比較した場合、双方のオペランドに対して strcmp(3C) を実行した場合と同じ結果になります。以下に、D の文字列比較とその結果の例を示します。

```
"coffee" < "espresso"          ... 1 (真) を返す
"coffee" == "coffee"           ... 1 (真) を返す
"coffee" >= "mocha"            ... 0 (偽) を返す
```

関係演算子は、列挙型のデータオブジェクトと、列挙によって定義された列挙子タグとの比較にも使用されます。列挙は、名前付きの整数定数を作成するときに使用します。詳細は、第 8 章「型と定数の定義」を参照してください。

論理演算子

D プログラムでは、次の二項論理演算子を使用できます。最初の 2 つの演算子は、対応する ANSI-C 演算子と同じ機能を持っています。

表 2–8 D の論理演算子

&&	論理積「*AND*」: 両方のオペランドが真の場合、真
||	論理和「*OR*」: いずれか一方のオペランド、または両方のオペランドが真の場合、真
^^	排他的論理和「*XOR*」: いずれか一方のオペランドだけが真の場合、真

論理演算子は、D の述語の記述によく使用されます。論理積 AND 演算子は、短絡評価を行います。 左側のオペランドが偽の場合、右側の式は評価されません。論理和

OR 演算子も短絡評価を行います。 左側のオペランドが真の場合、右側の式は評価されません。排他的論理和 XOR 演算子は、短絡評価を行いません。 必ず両方の式のオペランドが評価されます。

二項論理演算子のほかに、単項演算子 ! も使用できます。この単項演算子は、単一のオペランドの論理否定を実行します。 つまり、0 オペランドを 1 に、0 以外のオペランドを 0 に変換します。! と `== 0` は機能上同等ですが、D プログラムでは、慣例上、ブール値を表す整数を操作するときは !、ブール型以外の整数を操作するときは == 0 を使用します。

論理演算子は、整数型またはポインタ型のオペランドに適用されます。論理演算子は、ポインタオペランドを符号なしの整数値と見なします。D のすべての論理演算子、関係演算子の場合と同じく、0 以外の整数値を持つオペランドは真、整数値 0 を持つオペランドは偽になります。

ビット演算子

D では、次の二項演算子を使って、整数オペランドに含まれる個々のビットを操作できます。これらの演算子の効果は、ANSI-C の場合と同じです。

表 2–9 D のビット演算子

&	ビット単位の論理積 AND
|	ビット単位の論理和 OR
^	ビット単位の排他的論理和 XOR
<<	左のオペランドを、右のオペランドに指定されたビット数だけ左にシフトする
>>	左のオペランドを、右のオペランドに指定されたビット数だけ右にシフトする

二項演算子 & では、整数オペランドのビットを消去できます。二項演算子 | では、整数オペランドにビットを設定できます。二項演算子 ^ は、対応するオペランドビットが 1 つだけ設定されたビット位置に 1 を返します。

シフト演算子では、指定された整数オペランド内のビットを左右に移動できます。左シフトでは、結果の右側の空のビット位置にゼロが格納されます。符号なし整数オペランドを使った右シフトでは、結果の左側の空のビット位置にゼロが格納されます。符号付き整数オペランドを使った右シフトでは、左側の空のビット位置に符号ビットの値が格納されます。この操作を「算術シフト演算」と呼びます。

ビット数が負の値である場合や、左側のオペランドのビット数より大きい場合、整数値シフトの結果は未定義です。D プログラムのコンパイル時にこの条件が検出されると、D コンパイラからエラーメッセージが表示されます。

二項論理演算子のほかに、単項演算子 ~ も使用できます。この単項演算子は、単一のオペランドのビット単位の否定を実行します。 つまり、オペランド内の 0 ビットは 1 ビット、1 ビットは 0 ビットに変換されます。

代入演算子

D では、次の二項代入演算子を使って、D 変数を変更できます。変更できるのは、D の変数と配列だけです。D の代入演算子では、カーネルデータオブジェクトや定数は変更できません。代入演算子の効果は、ANSI-C の場合と同じです。

表 2–10　D の代入演算子

=	左のオペランドを右の式の値と等しくする
+=	左のオペランドを右の式の値の分だけ増分する
-=	左のオペランドを右の式の値の分だけ減分する
*=	左のオペランドに右の式の値を掛ける
/=	左のオペランドを右の式の値で割る
%=	左のオペランドを右の式の値で割ったときの剰余を求める
|=	左のオペランドと右の式の値の論理和 OR をビット単位で計算する
&=	左のオペランドと右の式の値の論理積 AND をビット単位で計算する
^=	左のオペランドと右の式の値の排他的論理和 XOR をビット単位で計算する
<<=	左のオペランドを、右の式の値で指定されたビット数だけ左にシフトする
>>=	左のオペランドを、右の式の値で指定されたビット数だけ右にシフトする

= 以外の代入演算子は、= とすでに説明したその他の演算子を組み合わせた演算の省略形として使用できます。たとえば、x = x + 1 という式は、x の数値が 1 度評価される点を除いて、x += 1 と同じです。これらの代入演算子は、以前に説明した二項のオペランドと同じ規則で使用します。

代入演算子の結果は、左側の式の新しい値と同等の式になります。代入演算子やこれまでに説明したその他の演算子を組み合わせると、複合式を表現できます。複合式の項は、丸括弧 ( ) を使ってグループ化できます。

インクリメント演算子とデクリメント演算子

D では、特殊な単項演算子 ++ と -- を使って、ポインタや整数の増分と減分を行うことができます。これらの演算子の効果は、ANSI-C の場合と同じです。これらの演算子は変数にのみ適用可能で、変数名の直前または直後に付加します。変数名の直前に演算子を付けた場合、まず変数が変更され、結果の式は変更後の変数の値と等しくなります。たとえば、次の 2 つの式の結果は同じです。

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

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

変数名の直後に演算子を付けた場合、式で使用する現在の値が返されたあと、変数が変更されます。たとえば、次の 2 つの式の結果は同じです。

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

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

インクリメント演算子とデクリメント演算子を使用すると、変数宣言なしで新しい変数を作成できます。変数宣言を省略して、変数にインクリメント演算子かデクリメント演算子を適用した場合、この変数の型は暗黙的に `int64_t` になります。

インクリメント演算子とデクリメント演算子は、整数変数またはポインタ変数に適用できます。これらの演算子を整数変数に適用した場合、対応する値が 1 増分または 1 減分されます。これらの演算子をポインタ変数に適用した場合、ポインタアドレスが、ポインタが参照しているデータ型のサイズの分だけ増分または減分されます。D のポインタとポインタ算術演算については、第 5 章「ポインタと配列」を参照してください。

条件式

D では if-then-else 構文はサポートされていませんが、演算子 ? と : を使った単純な条件式がサポートされています。これらの演算子では、3 つの式を関連付けることができます。具体的には、最初の式が、残りの 2 つの式のうちどちらを評価するかを決める、条件式になります。たとえば、次の D 文では、値 i に応じて、2 つの文字列のうちのどちらかが、変数 x の設定に使用されます

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

この例では、最初に式 `i == 0` が評価され、真であるか偽であるかが求められます。最初の式が真の場合、2 番目の式が評価され、?: 式の値が返されます。最初の式が偽の場合、3 番目の式が評価され、?: 式の値が返されます。

ほかの D 演算子の場合と同様に、複数の ?: 演算子を 1 つの式で使用することで、より複雑な式を作成できます。たとえば、次の式は、0 - 9、a - z、A - Z のいずれかの文字を含む char 変数 c を取り、この文字を数字と見なした場合の値を 16 進 (base 16) の整数として返します。

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

?: が使用されている最初の式の値が真と評価されるためには、ポインタまたは整数である必要があります。2 番目と 3 番目の式は、互換性のある型であれば何でもかまいません。ただし、一方が文字列型、もう一方が整数型を返すような条件式は作成できません。また、2 番目と 3 番目の式では、trace()、printf() などのトレース関数は呼び出せません。条件付きでデータをトレースしたい場合は、第 1 章「はじめに」で説明したように、述語を使用してください。

型変換

式に含まれる複数のオペランドの型が違っていても、相互に互換性のある型であれば、型変換が実行されて、結果の式の型が決まります。D の型変換の規則は、ANSI-C の整数の算術変換規則と同じです。この規則を、「通常の算術変換」と呼びます。

変換規則について簡単に説明します。 整数型はそれぞれ char、short、int、long、long long の順に順位付けられます。符号なし型は、対応する符号付き型より上位 (ただし 1 つ上位の整数型より下位) に順位付けられます。2 つの整数オペランドを使って x + y のような式を作成したとしましょう。この 2 つのオペランドの型が異なる場合、順位が高いほうの型が、式の結果の型になります。

変換が必要な状況では、順位が低いほうの型が、順位の高いほうの型に「拡張」されます。型が拡張されても、オペランドの値が変更されるわけではありません。 値は、その符号に従って、より大きいコンテナに拡張されるだけです。符号なしオペランドの拡張では、結果の整数の未使用の上位ビットにゼロが格納されます。符号付きオペランドの拡張では、未使用の上位ビットに符号拡張の実行結果が格納されます。符号付きの型を符号なしの型に変換した場合、最初に符号付きの型が拡張され、その後、変換によって求められた、新しい符号なしの型が割り当てられます。

整数などの型は、明示的に別の型に「キャスト」することもできます。D のポインタと整数は、任意の整数型またはポインタ型にキャストできます。ただし、これ以外の型にキャストすることはできません。文字列や文字配列のキャストと拡張の規則については、第 6 章「文字列」で説明します。整数またはポインタをキャストするときは、次のような式を使用します。

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

丸括弧の中に最終的な型を指定し、元となる式の前に付加します。整数をより上位の型にキャストするときは、拡張が行われます。整数をより下位の型にキャストするときは、不要な上位ビットにゼロが格納されます。

D では、浮動小数点算術演算は実行できません。そのため、浮動小数点オペランドの変換やキャストは許可されておらず、また、暗黙的な浮動小数点変換の規則も定義されていません。

優先度

以下の表に、D の演算子の優先順位と結合の規則を一覧します。これらの規則は少し複雑ですが、ANSI-C の演算子の優先順位の規則との完全な互換性を実現するためには欠かせないものです。表の項目は、上から優先順位の高い順になっています。

表 2–11　D の演算子の優先順位と結合規則

演算子	結合規則
`() [] -> .`	左から右
`! ~ ++ -- + - * & (`*type*`) sizeof stringof offsetof xlate`	右から左
`* / %`	左から右
`+ -`	左から右
`<< >>`	左から右
`< <= > >=`	左から右
`== !=`	左から右
`&`	左から右
`^`	左から右
`|`	左から右
`&&`	左から右
`^^`	左から右
`||`	左から右
`?:`	右から左
`= += -= *= /= %= &= ^= |= <<= >>=`	右から左
`,`	左から右

この表の中には、まだ説明していない演算子も含まれています。これらの演算子については、次の章以降で説明します。

`sizeof`	オブジェクトのサイズを計算する (第 7 章「構造体と共用体」)
`offsetof`	型メンバーのオフセットを計算する (第 7 章「構造体と共用体」)
`stringof`	オペランドを文字列に変換する (第 6 章「文字列」)
`xlate`	データ型を翻訳する (第 40 章「トランスレータ」)
単項の `&`	オブジェクトのアドレスを計算する (第 5 章「ポインタと配列」)
単項の `*`	オブジェクトへのポインタを間接参照する (第 5 章「ポインタと配列」)
`->` と `.`	構造体型または共用体型のメンバーにアクセスする (第 7 章「構造体と共用体」)

表中のコンマ (`,`) 演算子は、ANSI-C のコンマ演算子との互換性を実現するために用意されたものです。この演算子は、複数の式を左から順に評価し、一番右の式の値を返します。この演算子は C との完全な互換性を実現するためのものであり、通常は使用しません。

演算子の優先順位の表にある `()` は、関数呼び出しを表します。`printf()`、`trace()` などの関数呼び出しの例は、第 1 章「はじめに」に記載されています。D では、コンマを使って、関数の引数を列挙したり、連想配列のキーのリストを作成したりできます。このコンマはコンマ演算子とは別のもので、「左から右」の順で評価を行うとはかぎりません。D コンパイラが関数の引数を評価する順番や、連想配列のキーを評価する順番は、特に決まっていません。このため、複数の式、たとえば `i` と `i++` を組み合わせて使用するときは、相互の副作用に注意してください。

演算子の優先順位の表にある `[]` は、配列または連想配列の参照を表します。連想配列の例は、第 1 章「はじめに」に記載されています。「集積体」と呼ばれる特殊な連想配列については、第 9 章「集積体」で説明します。演算子 `[]` を使って、固定サイズの C 配列にインデックスを付けることもできます。これについては、第 5 章「ポインタと配列」を参照してください。

◆ ◆ ◆ 第 3 章

3 変数

D のトレースプログラムでは、使用できる基本型の変数が 2 種類あります。 スカラー変数と連想配列です。第 1 章の例では、これらの変数の使用方法の概要を説明しました。この章では、D 変数の規則の詳細と、変数をさまざまなスコープに関連付ける方法について説明します。「集積体」と呼ばれる特殊な配列変数については、第 9 章「集積体」で説明します。

スカラー変数

スカラー変数では、整数やポインタなど、個々の固定サイズのデータオブジェクトを表現できます。スカラー変数では、1 つ以上のプリミティブ型や複合型で構成された固定サイズのオブジェクトも表現できます。D では、オブジェクトの配列と複合構造体の両方を作成できます。DTrace では、事前定義された最大長まで文字列を伸ばすことによって、これを固定サイズのスカラーとして表現することもできます。D プログラム内の文字列の長さの制御については、第 6 章「文字列」で詳しく説明します。

スカラー変数は、D プログラム内の未定義の識別子に初めて値を割り当てたときに、自動的に作成されます。たとえば、`int` 型のスカラー変数 `x` は、任意のプローブ節内で `int` 型の値を割り当てるだけで作成できます。

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

このようにして作成されたスカラー変数は、「大域的な」変数です。 この変数の名前とデータ格納場所は、1 度定義されると、D プログラムのすべての節で認識されます。識別子 `x` を参照するたびに、この変数に関連付けられた単一の格納場所が参照されます。

D では、ANSI-C の場合とは違って、明示的変数宣言は必要ありません。大域変数を使用する前に、その名前と型を明示的に割り当てて、大域変数を宣言したい場合は、次の例のように、プログラム内のプローブ節の外側に宣言を挿入します。ほとんどの D プログラムは明示的変数宣言を必要としませんが、明示的変数宣言が便利な場合もあります。たとえば、変数の型を細かく制御したい場合や、プログラムの冒頭に一連の宣言とコメントを挿入して、プログラム内で使用する変数やその意味を記述したい場合などです。

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

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

D の変数宣言では、ANSI-C 宣言の場合とは違って、初期値の割り当てを行いません。初期値の割り当てには必ず、`BEGIN` プローブ節を使用します。すべての大域変数の記憶域には、最初にこの変数を参照する前に、DTrace によってゼロが格納されます。

D 言語定義では、D 変数のサイズと数は制限されていません。ただし、DTrace 実装とシステムの使用可能なメモリー容量による制限が存在します。こうした制限のうちプログラムのコンパイル時に適用可能なものは、D コンパイラによって適用されます。プログラムの制限関連のオプションの設定方法については、第 16 章「オプションとチューニング可能パラメータ」で詳しく説明します。

連想配列

連想配列では、「キー」と呼ばれる名前を指定して取得できるデータ要素の集積を表現できます。D の連想配列のキーは、「組」と呼ばれるスカラー式の値のリストで構成されます。配列の組は、配列の参照時に対応する配列値を取得するために呼び出される関数のパラメータリストのようなものです。D の連想配列には、それぞれ決まった型の、決まった数の組要素で構成される、固定の「キー署名」が 1 つずつ割り当てられています。D プログラムでは、配列ごとに異なるキー署名を定義できます。

連想配列は、通常の固定サイズの配列とは異なり、要素数があらかじめ制限されていません。単に整数をキーとして使用するのではなく、任意の組で要素にインデックスを付けることができます。また、要素は、事前に割り当てられた連続した記憶域に格納されるわけではありません。連想配列は、C、C++、Java™ 言語プログラム内の、ハッシュテーブルなどの単純な辞書データ構造を使用する場合に便利です。連想配列を使用すると、D プログラム内で捕捉したイベントや状態の動的な履歴を作成し、より複雑な制御フローを実現できます。

連想配列を定義するには、次のような形式の代入式を記述します。

name [*key*] = *expression* ;

name は任意の有効な D 識別子、*key* は 1 つ以上の式をコンマで区切った形式のリストです。たとえば、以下の式は、キー署名 [int, string] を持つ連想配列 a を定義し、[123, "hello"] という組で指定された場所に整数値 456 を格納します。

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

配列内の各オブジェクトの型は、その配列内のすべての要素に共通の型で、固定されています。たとえば、a には最初に整数 456 が割り当てられているので、その後この配列に格納される値はすべて int 型になります。連想配列の要素を変更するときは、第 2 章で定義した代入演算子を、それぞれに定義されたオペランド規則に従って使用します。非互換な割り当てがあると、D コンパイラからエラーメッセージが返されます。連想配列のキーや値には、スカラー変数で使用できる任意の型を使用できます。連想配列をキーまたは値として、ほかの連想配列内に入れ子にすることはできません。

連想配列は、配列キー署名と互換性のある任意の組を使って参照できます。組の互換性の規則は、関数呼び出しと変数割り当ての互換性の規則に似ています。 同じ長さの組を使用する必要があります。また、実パラメータリスト内の型と、正規のキー署名内の対応する型との間には、互換性がなければなりません。たとえば、連想配列 x を次のように定義したとします。

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

この場合、キー署名は unsigned long long 型、値は int 型になります。この配列は、式 x['a'] を使って参照することもできます。これは、59 ページの「型変換」で説明した算術変換規則により、長さが 1 の int 型の文字定数 'a' で構成された組と、unsigned long long 型のキー署名に互換性があるからです。

D の連想配列を明示的に宣言してから使用する必要がある場合は、プログラムソースコードのプローブ節の外側に、配列名とキー署名の宣言を記述します。

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

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

連想配列の定義後は、互換性のあるキー署名を持つ組がすべて参照可能になります。まだ割り当てられていない組も参照できます。まだ割り当てられていない連想配列要素にアクセスすると、定義により、ゼロが格納されたオブジェクトが返されます。この定義によると、連想配列要素に配下の記憶域を割り当てるためには、この要素にゼロ以外の値を割り当てる必要があります。反対に、連想配列要素にゼロ

を割り当てると、DTrace により、配下の記憶域の割り当てが解除されます。この動作は重要な意味を持っています。なぜなら、連想配列要素に割り当てられる動的な変数空間には限りがあるからです。割り当てる空間が足りないと、割り当てに失敗し、動的な変数の中断を示すエラーメッセージが表示されます。使用していない連想配列要素には、常にゼロを割り当ててください。動的な変数の中断を予防するその他のテクニックについては、第 16 章「オプションとチューニング可能パラメータ」を参照してください。

スレッド固有変数

DTrace では、個々のオペレーティングシステムスレッドに固有の変数記憶域を宣言できます。こうしたスレッド固有変数は、以前に説明した大域変数とは対照的な機能を備えています。スレッド固有変数は、プローブを有効にし、有効にしたプローブを起動するすべてのスレッドにタグやその他のデータでマークを付けたい場合に便利です。D では、このようなプログラムを簡単に作成できます。というのも、スレッド固有変数は、D コード内で共通の名前を使用して、個々のスレッドに関連付けられた別々のデータ記憶域を参照できるからです。スレッド固有変数を参照するには、特殊な識別子 `self` に、演算子 `->` を適用します。

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

この D コードの抜粋では、read(2) システムコールの呼び出し時にプローブが有効になり、このプローブを起動する各スレッドに、`read` という名前のスレッド固有変数が関連付けられます。スレッド固有変数は、大域変数と同じく、最初の割り当て時に自動的に作成されます。また、最初の代入文の右式で使用されている型 (この例では `int`) が適用されます。

D プログラム内で変数 `self->read` が参照されるたびに、対応する DTrace プローブが起動したときに実行されていたオペレーティングシステムスレッドに関連付けられたデータオブジェクトが参照されます。スレッド固有変数は、システム内のスレッドの識別情報を示す組によって暗黙的なインデックスが付けられた、連想配列と見なすことができます。スレッドの識別情報は、システムが実行している間は変わりません。 あるスレッドを終了したあと、同じオペレーティングシステムのデータ構造で新しいスレッドを生成した場合、DTrace スレッド固有記憶域の識別情報は再利用されません。

スレッド固有変数の定義後は、この変数を使って、システム内の任意のスレッドを参照できます。これは、この変数が特定のスレッドに割り当てられていない場合も同様です。スレッド固有変数のスレッドのコピーがまだ割り当てられていない場合は、コピーのデータ記憶域にゼロが格納されて定義されます。連想配列要素の場合と同じく、スレッド固有変数に配下の記憶域を割り当てるには、この変数にゼロ以

外の値を割り当てる必要があります。また、連想配列要素の場合と同じく、スレッド固有変数にゼロを割り当てると、DTrace により、配下の記憶域の割り当てが解除されます。使用していないスレッド固有変数には、常にゼロを割り当ててください。スレッド固有変数に割り当てる動的な変数空間を微調整するその他のテクニックについては、第 16 章「オプションとチューニング可能パラメータ」を参照してください。

D プログラムでは、連想配列をはじめとする任意の型のスレッド固有変数を定義できます。以下に、スレッド固有変数の定義例を示します。

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

スレッド固有変数は、ほかの D 変数の場合と同じく、明示的変数宣言なしで使用できます。あえて宣言を作成する場合は、キーワード `self` を前に付けて、プログラム節の外側に配置します。

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

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

スレッド固有変数は、大域変数とは別の名前空間に格納されるので、変数名を再利用できます。プログラム内で名前を多重定義した場合、`x` と `self->x` はそれぞれ別の変数になります。以下に、スレッド固有変数の使用例を示します。テキストエディタで以下のプログラムを入力し、`rtime.d` という名前のファイルに保存してください。

例 3–1　`rtime.d`: `read(2)` の実行開始からの経過時間を計算する

```
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.
```

例 3–1 rtime.d: read(2) の実行開始からの経過時間を計算する (続き)

```
     */
    self->t = 0;
}
```

では、シェルに戻ってプログラムを実行しましょう。しばらくすると、結果が出力されます。何も出力されない場合は、いくつかのコマンドを実行してみてください。

```
# 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
#
```

rtime.d は、スレッド固有変数 t を使って、スレッドが read(2) に入ったときのタイムスタンプを捕捉します。続いて、戻り節で、現在のタイムスタンプと self->t の差分が計算され、read(2) の実行が開始されてからの経過時間が出力されます。組み込み D 変数 pid と tid は、read(2) の実行スレッドのプロセス ID とスレッド ID を報告します。この情報を得たあと、不要になった self->t には 0 が割り当てられます。このため、DTrace は、t に関連付けられた配下の記憶域を現在のスレッドで再利用できます。

通常、read(2) は、サーバープロセスとデーモンによってバックグラウンドで実行されているので、ユーザーが操作を停止している間も出力行が続々と表示されていきます。execname 変数を使って read(2) の実行プロセス名を出力し、よりわかりやすい情報を得たい場合は、rtime.d の 2 番目の節を次のように変更します。

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

調べたいプロセスが見つかったら、述語を追加して、read(2) の動作を詳しく調べることができます。

```
syscall::read:entry
/execname == "Xsun"/
{
```

```
    self->t = timestamp;
}
```

節固有変数

D プログラムの節ごとに再利用可能な記憶域を持つ D 変数も定義できます。節固有変数は、C、C++、Java 言語プログラムの自動変数とよく似ており、1 回の関数呼び出しが完了するまで有効です。節固有変数は、すべての D プログラム変数と同じく、最初の割り当て時に作成されます。これらの変数を参照し、割り当てるには、特殊な識別子 `this` に、演算子 `->` を適用します。

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

節固有変数を明示的に宣言してから使用したい場合は、キーワード `this` を使用します。

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

BEGIN
{
    this->x = 123;
    this->c = 'D';
}
```

節固有変数は、プローブ節の終了まで有効です。DTrace により、プローブ節に関連付けられたアクションの実行が完了すると、すべての節固有変数用の記憶域が回収され、次の節で再利用されます。したがって、節固有変数は、D 変数の中で唯一、ゼロに初期化されることがありません。プログラム内の単一のプローブに複数の節が含まれている場合、これらの節の実行が完了するまで、すべての節固有変数はそのままの状態を保持します。次の例を参照してください。

例 3–2 `clause.d`: 節固有変数

```
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.
     */
```

例 3–2 clause.d: 節固有変数 (続き)

```
    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++);
}
```

節は常に記述された順に実行され、節固有変数は同じプローブを有効にする複数の節を通して変更されないので、上記のプログラムを実行したときの出力は、いつも同じになります。

```
# 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
```

節固有変数は、同じプローブを有効にする複数の節を通して変更されないので、あるプローブに対して実行される最初の節では、その値は未定義になります。節固有変数を使用するときは、必ず事前に適切な値を割り当ててください。そうしないと、プログラムの結果が予想どおりにならない可能性があります。

節固有変数は、任意のスカラー変数型を使って定義できますが、連想配列は、節固有のスコープでは定義できません。節固有変数のスコープは、対応する変数データだけに適用されます。この変数に定義された名前や型の識別情報には適用されません。節固有変数の定義後は、以降のすべての D プログラム節で、この名前と型署名を使用できます。記憶域は、節が変わると変更される可能性があります。

節固有変数を使って、計算の途中結果を累積できます。また、節固有変数をほかの変数の一時コピーとして使用することもできます。節固有変数には、連想配列よりも高速にアクセスできます。そのため、同じ D プログラム節内で特定の連想配列値を繰り返し参照する必要がある場合は、この値を節固有変数にコピーし、節固有変数を繰り返し参照するようにすると、効率がよくなります。

組み込み変数

以下の表に、D の組み込み変数を一覧します。これらの変数は、すべてスカラー大域変数です。現在、D では、スレッド固有変数、節固有変数、および組み込みの連想配列は定義されていません。

表 3–1　DTrace の組み込み変数

型と名前	説明
`int64_t arg0、...、arg9`	プローブに渡される最初の 10 個の入力引数は、生の 64 ビット整数として表現されます。プローブに渡された引数の数が 10 個未満の場合、残りの変数はゼロを返します。
`args[]`	現在のプローブに渡される型付き引数です (存在する場合)。`args[]` 配列へのアクセスには整数インデックスが使用されますが、各要素には、指定のプローブ引数に対応する型が定義されます。たとえば、read(2) システムコールプローブで `args[]` を参照する場合、`args[0]` の型は `int`、`args[1]` の型は `void *`、`args[2]` の型は `size_t` になります。
`uintptr_t caller`	現在のプローブを入力する直前の、現在のスレッドのプログラムカウンタの場所を示します。
`chipid_t chip`	現在の物理チップの CPU チップ識別子です。詳細は、第 26 章「sched プロバイダ」を参照してください。
`processorid_t cpu`	現在の CPU の CPU 識別子です。詳細は、第 26 章「sched プロバイダ」を参照してください。

表 3–1 DTrace の組み込み変数 (続き)

型と名前	説明
`cpuinfo_t *curcpu`	現在の CPU の CPU 情報です。詳細は、第 26 章「sched プロバイダ」を参照してください。
`lwpsinfo_t *curlwpsinfo`	現在のスレッドに関連付けられている軽量プロセス (LWP) の LWP 状態です。この構造の詳細は、proc(4) のマニュアルページに記載されています。
`psinfo_t *curpsinfo`	現在のスレッドに関連付けられているプロセスのプロセス状態です。この構造の詳細は、proc(4) のマニュアルページに記載されています。
`kthread_t *curthread`	現在のスレッドのオペレーティングシステムカーネルの内部データ構造 `kthread_t` のアドレスです。`kthread_t` は `<sys/thread.h>` に定義されています。この変数とその他のオペレーティングシステムデータ構造の詳細は、『SOLARIS インターナル — カーネル構造のすべて』を参照してください。
`string cwd`	現在のスレッドに関連付けられているプロセスの現在の作業ディレクトリ名です。
`uint_t epid`	現在のプローブの有効なプローブ ID (EPID) です。この整数は、特定の述語と一連のアクションによって有効化された特定のプローブを一意に識別します。
`int errno`	このスレッドによって直前に実行されたシステムコールから返されるエラー値です。
`string execname`	現在のプロセスを実行するため、exec(2) に渡された名前です。
`gid_t gid`	現在のプロセスの実グループ ID です。
`uint_t id`	現在のプローブのプローブ ID です。この ID は、DTrace によって発行された、システム内のプローブを一意に識別する識別子です。この ID を確認するには、`dtrace -l` を実行します。
`uint_t ipl`	現在の CPU でのプローブ起動時の割り込み優先レベル (IPL) を表します。Solaris オペレーティングシステムカーネルでの割り込みレベルと割り込み処理の詳細は、『SOLARIS インターナル — カーネル構造のすべて』を参照してください。

表 3–1 DTrace の組み込み変数 (続き)

型と名前	説明
lgrp_id_t lgrp	現在の CPU をメンバーに持つ応答時間グループの応答時間グループ ID です。詳細は、第 26 章「sched プロバイダ」を参照してください。
pid_t pid	現在のプロセスのプロセス ID です。
pid_t ppid	現在のプロセスの親プロセスのプロセス ID です。
string probefunc	現在のプローブの記述に含まれる関数名の部分です。
string probemod	現在のプローブの記述に含まれるモジュール名の部分です。
string probename	現在のプローブの記述に含まれる名前の部分です。
string probeprov	現在のプローブの記述に含まれるプロバイダ名の部分です。
psetid_t pset	現在の CPU が含まれているプロセッサセットのプロセッサセット ID です。詳細は、第 26 章「sched プロバイダ」を参照してください。
string root	現在のスレッドに関連付けられているプロセスのルートディレクトリ名です。
uint_t stackdepth	現在のスレッドのプローブ起動時のスタックフレームの深さを表します。
id_t tid	現在のスレッドのスレッド ID です。スレッドがユーザープロセスに関連付けられている場合、この値は、pthread_self(3C) の呼び出し結果と同じになります。
uint64_t timestamp	ナノ秒タイムスタンプカウンタの現在の値です。このカウンタの値は、過去の任意の時点から増分しています。そのため、このカウンタは、相対計算専用です。
uid_t uid	現在のプロセスの実ユーザー ID です。
uint64_t uregs[]	現在のスレッドの、プローブ起動時のユーザーモード登録値 (保存済み) です。uregs[] 配列の使用方法については、第 33 章「ユーザープロセスのトレース」を参照してください。

表 3–1 DTrace の組み込み変数 (続き)

型と名前	説明
`uint64_t vtimestamp`	ナノ秒タイムスタンプカウンタの現在の値です。CPU 上で現在のスレッドの実行が開始されてから現在までの経過時間から、DTrace の述語およびアクションで使用された時間を減じた値になります。このカウンタの値は、過去の任意の時点から増分しています。そのため、このカウンタは、相対時間計算専用です。
`uint64_t walltimestamp`	1970 年 1 月 1 日の協定世界時 00:00 から現在までの経過時間をナノ秒単位で示します。

`trace()` をはじめとする D 言語の組み込み関数については、第 10 章「アクションとサブルーチン」で説明します。

外部変数

D では、オペレーティングシステムには定義されているが D プログラムには定義されていない変数にアクセスする際、特殊なスコープ演算子として、逆引用符 (‘) を使用します。たとえば、Solaris カーネルには、メモリーアロケータのデバッグ機能を有効にする、チューニング可能なシステム変数 `kmem_flags` の C 宣言が含まれています。`kmem_flags` の詳細は、『Solaris カーネルのチューンアップ・リファレンスマニュアル』を参照してください。このチューニング可能な変数は、次のように、カーネルソースコード内に C 変数として宣言されています。

```
int kmem_flags;
```

D プログラム内でこの変数の値にアクセスするには、次のような D 表記法を使用します。

```
‘kmem_flags
```

DTrace では、各カーネルシンボルと、対応するオペレーティングシステムの C コード内のシンボルの型が関連付けられます。このため、ソースからネイティブのオペレーティングシステムデータ構造に簡単にアクセスできます。オペレーティングシステムの外部変数を使用するには、対応するオペレーティングシステムのソースコードにアクセスする必要があります。

D プログラムから外部変数へのアクセスは、別のプログラム (オペレーティングシステムカーネルやそのデバイスドライバなど) の内部実装の詳細へのアクセスを意味します。このように、実装の詳細にアクセスする場合、信頼性の高い安定したインタフェースは提供されません。こうした詳細に依存する D プログラムは、ソフトウェアの該当部分をアップグレードすると、動作しなくなる可能性があります。こ

のため、外部変数は、通常、カーネルおよびデバイスドライバの開発者やサービス担当者が、DTrace を使ってパフォーマンスや機能上の問題のデバッグを行うときに使用されます。D プログラムの安定性の詳細は、第 39 章「安定性」を参照してください。

カーネルシンボル名は、D の変数や関数の識別子とは別の名前空間に格納されます。したがって、カーネルシンボル名と D 変数名の競合が発生することはありません。変数の直前に逆引用符を付けると、D コンパイラは、一致する変数定義を見つけるため、ロードされたモジュールのリストの順番に従って、既知のカーネルシンボルを検索します。Solaris カーネルは、動的にロードされたモジュールごとに独立したシンボル用名前空間を提供します。このため、有効なオペレーティングシステムカーネル内で同じ変数名を繰り返し使用できます。名前の競合を防ぐには、シンボル名の逆引用符より先に、アクセスされる必要のある変数を持つカーネルモジュール名を指定します。たとえば、ロード可能なカーネルモジュールは、通常、`_fini(9E)` 関数を提供します。したがって、`foo` という名前のカーネルモジュールが提供する `_fini` 関数のアドレスを参照するには、次のように記述します。

```
foo`_fini
```

外部変数には、オペランドの型の通常規則に従って、値を変更するものは除く、任意の D 演算子を適用できます。DTrace を起動すると、D コンパイラにより、アクティブなカーネルモジュールに対応する変数名のセットがロードされます。このため、これらの変数の宣言は不要です。外部変数には、値を変更するような演算子 (=、+= など) を適用してはいけません。DTrace では、安全性の面から、監視対象のソフトウェアの状態を改変することは禁じられています。

◆ ◆ ◆ 第 4 章

D プログラムの構造

D プログラムは、有効にするプローブ、およびプローブに結合する述語とアクションについて説明する、一連の節で構成されています。D プログラムには、第 3 章「変数」で説明した変数宣言や、第 8 章「型と定数の定義」で説明する新しい型の定義も含めることができます。この章では、D プログラムの全体構造と、複数のプローブを表すプローブ記述の作成機能について説明します。また、D プログラムでの C プリプロセッサ cpp の使用についても説明します。

プローブ節と宣言

これまでの例からわかるように、D プログラムのソースファイルは、DTrace によって有効化される計測機能について記述した 1 つ以上のプローブ節で構成されます。各プローブ節の標準形式は、次のとおりです。

```
probe descriptions
/ predicate /
{
   action statements
}
```

述語とアクション文のリストは省略可能です。プローブ節の外側に置かれた指令を「宣言」と呼びます。宣言は、必ずプローブ節の外側に挿入します。宣言を中括弧 ({ }) 内に記述することはできません。また、宣言を上記のプローブ節の要素と要素の間に配置することはできません。D プログラムの要素間の区切りと、アクション文のインデントには、空白を使用します。

宣言では、D 変数や外部 C シンボルを宣言したり (第 3 章「変数」を参照)、D プログラムで使用する新しい型を定義したり (第 8 章「型と定数の定義」を参照) できます。D プログラムでは、「プラグマ」と呼ばれる特殊な D コンパイラ指令も使用できます。プローブ節の外側も含めた任意の場所で使用できます。D プラグマは、先

頭文字が # の行に指定されます。D プラグマは、実行時 DTrace オプションの設定などに使用されます。詳細は、第 16 章「オプションとチューニング可能パラメータ」を参照してください。

プローブ記述

D プログラム節は、1 つ以上のプローブ記述で始まります。プローブ記述は、通常、以下の形式をとります。

provider:*module*:*function*:*name*

プローブ記述の一部のフィールドを省略した場合、D コンパイラは、指定されたフィールドだけを右から左の順で解釈します。たとえば、`foo:bar` というプローブ記述は、関数 `foo` を持つ `bar` という名前のプローブを表します。プローブのプロバイダとモジュールのフィールドの値は無視されます。このため、プローブ記述は、名前に基づいて 1 つ以上のプローブと照合される「パターン」と見なすことができます。

D プローブ記述を作成するときは、左側に適切な「プロバイダ」を指定できるように、4 つのフィールドの区切り記号をすべて指定してください。プロバイダを指定しないと、複数のプロバイダが同じ名前のプローブを発行している場合に、予想外の結果になる可能性があります。同様に、プローブ記述の一部を省略していると、将来のバージョンの DTrace で追加された新しいプロバイダのプローブが、予期せずして照合されてしまう可能性もあります。プロバイダを指定して、モジュール、関数、名前のフィールドを未指定にした場合、このプロバイダのすべてのプローブが照合されます。たとえば、DTrace の `syscall` プロバイダから発行されたすべてのプローブを表現するには、`syscall:::` のように記述します。

プローブ記述では、シェルの「展開」パターンマッチング構文 (`sh(1)` を参照) とよく似たパターンマッチング構文も使用できます。記述からプローブを照合する前に、DTrace により、各記述フィールドに *、?、および [の文字がないか検査されます。プローブ記述フィールドに、このうちいずれかの文字が含まれていて、その直前の文字が \ でない場合、このフィールドはパターンと見なされます。記述パターンは、プローブの対応するフィールド全体と一致する必要があります。完全な形のプローブ記述では、プローブ記述のすべてのフィールドが一致して初めて、プローブが照合され有効化されます。パターンが使用されていないプローブ記述フィールドは、対応するプローブのフィールドに完全に一致する必要があります。空の記述フィールドは、すべてのプローブを表します。

以下の表に、プローブ名のパターン内で認識される特殊文字を示します。

表 4–1 プローブ名のパターンマッチング文字

記号	説明
*	NULL 文字列を含むすべての文字列を表します。
?	任意の 1 文字を表します。
[...]	括弧内のいずれか 1 文字を表します。2 つの文字の間に - を入力すると、その範囲 (両端の文字を含む) の任意の文字を表します。[に続く最初の文字を ! にすると、括弧内に含まれない任意の文字を表します。
\	次の 1 文字に特別な意味を持たせず、文字どおりに解釈します。

パターンマッチング文字は、プローブ記述の 4 つのフィールド全部、またはそのいずれかで使用できます。`dtrace -l` とともにコマンド行にパターンを入力して、条件に一致するプローブを一覧することもできます。たとえば、`dtrace -l -f kmem_*` というコマンドでは、DTrace プローブのうち、関数の名前が `kmem_` で始まるものすべてが一覧されます。

複数のプローブ記述または記述パターンで同じ述語とアクションを指定したい場合は、コンマで区切ったリストの形式で記述します。たとえば次の D プログラムでは、「`lwp`」または「`sock`」という語を含むシステムコールの開始に関連付けられたプローブが起動するたびに、タイムスタンプがトレースされます。

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

プローブ記述では、整数のプローブ ID を使って、プローブを指定することもできます。たとえば、次の節を参照してください。

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

この節では、`dtrace -l -i 12345` で報告される、プローブ ID 12345 を有効にすることができます。D プログラムを作成するときは、常に、人間が読める形式のプローブ記述を使用する必要があります。整数のプローブ ID は、DTrace プロバイダカーネルモジュールのロード後、アンロード後、または再起動後に変更される可能性があります。

述語

述語は、スラッシュ (//) で囲まれた式であり、プローブ起動時に評価され、関連アクションを実行するかどうかを決定します。述語は、D プログラムでより複雑な制御フローを構築するために使用する主要な条件構文です。プローブ節の述語部分を完全に省略することもできます。この場合、プローブの起動時に常にアクションが実行されることになります。

述語式では、これまでに紹介したすべての D 演算子を使用できます。また、変数や定数を含むあらゆる D データオブジェクトを参照できます。述語式は、真または偽の結果を導き出すため、整数型またはポインタ型の値を評価する必要があります。すべての D 式と同じく、ゼロ値は偽、ゼロ以外の値は真と解釈されます。

アクション

プローブのアクションは、セミコロン (;) で区切られた文のリストを中括弧 ({ }) で囲んだ形式で記述します。データのトレースなどのアクションの実行を省略して、特定の CPU 上で特定のプローブが起動したという事実だけを記録したい場合は、中に何も文が記述されていない、空の中括弧を指定します。

C プリプロセッサの使用

Solaris システムインタフェースの定義に使用されている C プログラミング言語には、C プログラムのコンパイルの一連の初期手順を実行する「プリプロセッサ」が用意されています。C プリプロセッサは、一般に、マクロ置換 (C プログラム内のあるトークンを別の事前定義済みトークンセットで置き換える) の定義や、システムヘッダーファイルのコピーの挿入に利用されます。`dtrace` コマンドに `-C` オプションを指定すると、D プログラムで C プリプロセッサを使用することができます。このオプションが指定された場合、`dtrace` はまず、プログラムソースファイルに対して cpp(1) プリプロセッサを実行します。そして、その結果を D コンパイラに渡します。C プリプロセッサの詳細は、『プログラミング言語 C』を参照してください。

D コンパイラには、オペレーティングシステム実装に関連付けられている C の型記述のセットが自動的にロードされます。しかし、プリプロセッサを使用すると、独自の C プログラムで使用している型定義などもロードできます。プリプロセッサではこのほかに、D コードのチャンクに拡張されるマクロやその他のプログラム要素を作成するなどのタスクも実行できます。D プログラムでプリプロセッサを使用するときは、有効な D 宣言を含むファイルだけを使用してください。型とシンボルの外部宣言だけを含む通常の C ヘッダーファイルは、D コンパイラで正しく解釈されます。C 関数ソースコードのような追加プログラム要素を含む C ヘッダーファイルは、D コンパイラでは構文解析できません。このようなファイルを使用した場合、D コンパイラからエラーメッセージが返されます。

◆ ◆ ◆ 第 5 章

5 ポインタと配列

ポインタは、オペレーティングシステムカーネル内またはユーザープロセスのアドレス空間内のデータオブジェクトのメモリーアドレスです。D では、ポインタを作成、操作して、変数や連想配列内に格納できます。この章では、ポインタの D 構文、ポインタの作成やアクセスに使用する演算子、およびポインタと固定サイズのスカラー配列との関係について説明します。また、異なるアドレス空間でのポインタの使用についても説明します。

注 – D ポインタ構文は、対応する ANSI-C 構文と同じです。C や C++ のプログラミング経験をお持ちのユーザーは、この章は、ざっと目を通すだけでかまいません。ただし、88 ページの「DTrace オブジェクトのポインタ」と88 ページの「ポインタとアドレス空間」は目を通すようにしてください。DTrace に固有の機能や問題が説明されています。

ポインタとアドレス

Solaris オペレーティングシステムでは、「仮想メモリー」により、ユーザープロセスごとに、システム上のメモリーリソースについて固有の仮想表示が提供されます。メモリーリソースの仮想表示を「アドレス空間」と呼びます。アドレス空間は、アドレス範囲 (32 ビットの場合 `[0 ... 0xffffffff]`、64 ビットの場合 `[0 ... 0xffffffffffffffff]`) と、翻訳セットを関連付けます。翻訳セットは、オペレーティングシステムやハードウェアが個々の仮想アドレスを対応する物理メモリー配置に変換するために使用されます。D のポインタはデータオブジェクトで、整数型の仮想アドレス値を格納したあと、対応するメモリー配置に格納されているデータの形式を説明する D 型と関連付けます。

D 変数をポインタ型として宣言するには、参照されるデータの型を指定し、型名にポインタ型の宣言であることを示すアスタリスク (*) を付加します。たとえば次のような宣言があるとします。

```
int *p;
```

この宣言は、p という名前の D 大域変数が、整数のポインタであることを示しています。この宣言から、p 自体が 32 ビットまたは 64 ビットの整数で、その値はメモリー内にある別の整数のアドレスであることがわかります。D コードは、コンパイル後、オペレーティングシステムカーネル内でプローブ起動時に実行されます。このため、D ポインタは通常、カーネルのアドレス空間に関連付けられています。稼働中のオペレーティングシステムカーネルのポインタのビット数を確認するには、`isainfo(1)` `-b` コマンドを実行します。

カーネル内のデータオブジェクトのポインタを作成したい場合は、& 演算子を使って、そのアドレスを計算します。たとえば、オペレーティングシステムカーネルのソースコードに、チューニング可能なパラメータ `int kmem_flags` が宣言されています。この `int` のアドレスをトレースするには、D でのオブジェクト名に & 演算子を付加した結果をトレースします。

```
trace(&‘kmem_flags);
```

* 演算子は、ポインタによってアドレス指定されたオブジェクトを参照するときに使用します。この演算子は、& 演算子と正反対の機能を持っています。たとえば、次の 2 つの D コードの抜粋は、意味的に同じです。

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

左側の抜粋コードでは、D 大域変数ポインタ p が作成されています。`kmem_flags` は `int` 型のオブジェクトなので、`&‘kmem_flags` の結果の型は `int *` (`int` へのポインタ) になります。左側の抜粋コードでは、ポインタをデータオブジェクト `kmem_flags` に移してから、`*p` の値をトレースしています。したがって、この抜粋コードは、データオブジェクトの名前を指定して直接その値をトレースする右側の抜粋コードと同じになります。

ポインタの安全性

C や C++ の知識をお持ちのユーザーは、前の節をお読みになって、ポインタの使い方を間違えるとプログラムがクラッシュするのではないかと、少し不安になったかもしれません。DTrace は堅牢で安全な環境であるため、このような間違いのある D プログラムを実行しても、プログラムがクラッシュする心配はありません。作成した D プログラムにバグがあっても、無効な D ポインタアクセスが原因で DTrace やオペレーティングシステムカーネルに障害やクラッシュが起きることはありません。DTrace は、無効なポインタアクセスを検出すると、計測機能を無効にし、デバッグに役立つ情報を報告します。

Java プログラミングの経験をお持ちであればご存知でしょうが、Java 言語では、このような安全面の理由から、ポインタはサポートされていません。D では、C で記述されたオペレーティングシステムの実装の本質的な部分として、ポインタを使用する

必要があります。しかし、DTrace には、Java プログラミング言語と同様の保護機構が実装されているため、プログラムにバグがあっても、そのプログラム自体やその他のプログラムに危害を与えることはありません。DTrace のエラー報告機能は、Java プログラミング言語の実行環境と同じように、プログラミングエラーを検出し、例外を報告します。

以下では、DTrace のエラー処理およびエラー報告機能について確認するため、ポインタを使って、意図的に不正な D プログラムを作成してみましょう。エディタで以下の D プログラムを入力し、`badptr.d` という名前のファイルに保存してください。

例 5–1 `badptr.d`: DTrace のエラー処理機能のデモ

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

`badptr.d` プログラムで作成されている D ポインタ `x` は、`int` へのポインタです。このポインタには、特殊かつ無効なポインタ値 `NULL` が割り当てられていますが、この値はアドレス 0 を表す組み込みの別名です。慣例上、アドレス 0 は常に無効と定義されます。このため、C プログラムや D プログラムでは、`NULL` は標識値として使用できます。このプログラムでは、キャスト式により、`NULL` が整数ポインタに変換されています。このポインタは式 `*x` によって間接参照され、その結果が別の変数 `y` に割り当てられます。その後、この変数 `y` のトレースが試みられます。この D プログラムを実行すると、`y = *x` という文が実行されたところで無効なポインタアクセスが検出され、エラーが報告されます。

```
# 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
#
```

プログラムで無効なポインタを使用した場合、「配置エラー」の問題が起きることもあります。構造上、整数をはじめとする基本データオブジェクトは、そのサイズに従ってメモリー内に配置されます。たとえば、2 バイトの整数は 2 の倍数のアドレス、4 バイトの整数は 4 の倍数のアドレスというように配置されます。4 バイトの整数のポインタを間接参照するとき、このポインタが 4 の倍数以外の無効な値のアドレスに配置されていると、アクセスに失敗し、配置エラーが返されます。たいていの場合、D の配置エラーは、D プログラム内のバグによってポインタの値が無効になったか、壊れていることを示します。配置エラーの例として、ソースコード

badptr.d で、NULL の代わりにアドレス (int *)2 を指定してみてください。int は 4 バイトで、2 は 4 の倍数ではないので、式 *x は DTrace 配置エラーになります。

DTrace のエラー機構の詳細は、199 ページの「ERROR プローブ」を参照してください。

配列宣言と記憶域

D では、第 3 章で説明した動的連想配列のほかに、「スカラー配列」もサポートされています。スカラー配列は、それぞれに同じ型の値が格納される、固定長の連続するメモリー配 置の集まりです。スカラー配列にアクセスするには、ゼロで始まる整数を使って、それぞれの位置を参照します。スカラー配列と C や C++ の配列の概念と構文は、直接対応しています。D では、スカラー配列は、連想配列やその応用である「集積体」ほどは多用されませんが、C で宣言された既存のオペレーティングシステムの配列データ構造にアクセスするとき、スカラー配列が必要になる場合があります。集積体については、第 9 章「集積体」で説明します。

以下では、int 型の 5 つの整数から成る D スカラー配列を宣言します。宣言の末尾には、接尾辞として、要素数を角括弧で囲んだものを付加します。

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

この配列の記憶域を視覚的に表現すると、次の図のようになります。

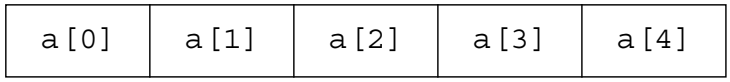

図 5–1 スカラー配列

D 式 a[0] は最初の配列要素、a[1] は 2 番目の配列要素 (以下同様) を参照しています。構文だけ見ると、スカラー配列は連想配列と非常によく似ています。たとえば、単一の整数キーによって参照される 5 つの整数から成る連想配列は、次のように宣言できます。

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

この配列は、式 a[0] を使って参照することもできます。一方、記憶域と実装について見ると、スカラー配列と連想配列は、まったく別物です。静的配列 a は、ゼロから順に番号付けされた連続した 5 つのメモリー配置で構成されています。インデックスは、この配列に割り当てられた記憶域内のオフセットを参照しています。これに対して、連想配列の場合、サイズは事前定義されておらず、要素は連続したメモリー配置には格納されません。また、連想配列のキーと対応する値の記憶域の位置には、何の関連性もありません。連想配列の要素 a[0] と a[-5] にアクセスする

と、DTrace により 2 ワード分のみの記憶域が割り当てられますが、これらの記憶域は連続しているとはかぎりません。連想配列のキーは、対応する値の抽象名になっており、値の記憶域の位置とは無関係です。

配列の作成時に初期値を割り当て、単一の整数式を配列インデックスとして使用した場合 (例: `a[0] = 2`)、式だけ見れば `a` をスカラー配列への代入と見なすことも可能ですが、D コンパイラは必ず、新しい連想配列を生成します。この場合、D コンパイラにこの配列をスカラー配列と判断させるには、あらかじめスカラー配列を宣言して、コンパイラに配列のサイズの定義を認識させる必要があります。

ポインタと配列の関係

D でも、ANSI-C の場合と同様に、ポインタと配列には特別な関係があります。配列を表す変数は、最初の記憶域の位置を示すアドレスに関連付けられています。ポインタも、記憶域の位置を示すアドレスであり、あらかじめ型定義されています。このため、D では、ポインタ変数でも配列変数でも、配列インデックスの表記 [ ] を使用できます。たとえば、次の 2 つの D コードの抜粋は、意味的に同じです。

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

左側の抜粋コードでは、式 `a[0]` に `&` 演算子を適用することにより、ポインタ `p` を `a` 内の最初の配列要素のアドレスに割り当てています。式 `p[2]` では、3 番目の配列要素 (インデックス 2) の値をトレースします。`p` には、`a` に関連付けられたアドレスが含まれるため、この式からは、右側の抜粋コードの `a[2]` と同じ値が導き出されます。このような等価性により、C と D では、任意のポインタまたは配列の任意のインデックスへのアクセスが可能です。コンパイラも DTrace 実行環境も、配列境界チェックは行いません。配列に事前定義された値の範囲外のメモリーにアクセスすると、予想外の結果になるか、先ほどの例のように、DTrace から、アドレスが無効であることを示すエラーが報告されます。DTrace そのものやオペレーティングシステムに影響はありませんが、D プログラムをデバッグする必要があります。

ポインタと配列の違いは、ポインタ変数が別の記憶域の整数アドレスを含む記憶域を参照する点です。配列変数は、配列の位置を含む整数アドレスではなく、配列の記憶域そのものを指定します。以下に、この違いを図示します。

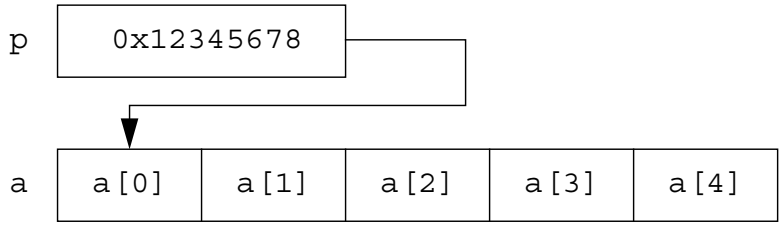

図5–2 ポインタと配列の記憶域

この違いは、ポインタを割り当てるときとスカラー配列を割り当てるときのD構文から明らかです。xとyがポインタ変数である場合、式x = yは正当です。この式では、y内のポインタアドレスが、xで指定された記憶域の位置にコピーされます。xとyがスカラー変数である場合、式x = yは不正です。Dでは配列そのものを割り当てることはできません。ただし、配列変数やシンボル名は、ポインタが可能なあらゆる状況で使用できます。pがポインタでaが配列である場合、p = aという文は使用可能です。この文は、p = &a[0]と同等です。

ポインタ演算

ポインタは、メモリー内の別のオブジェクトのアドレスとして使用される整数に過ぎません。このため、Dには、ポインタ演算機能が用意されています。ただし、ポインタ演算は、整数の演算とは別のものです。ポインタ演算では、ポインタで参照されている型のサイズとオペランドの乗除によって、配下のアドレスが暗黙的に変更されます。以下に、この属性を示すDコードの抜粋を示します。

```
int *x;

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

このコードの抜粋では、整数ポインタxが作成され、その値とその値を1増分した値、さらにその値を2増分した値がトレースされます。このプログラムを作成して実行したとき、DTraceから返される整数値は、0、4、8です。

xは整数(サイズは4バイト)のポインタなので、xの値を1増分すると、配下のポインタ値は4大きくなります。この特性は、ポインタを使って配列などの連続した記憶域の位置を参照する場合に便利です。たとえば、xを図5–2のような配列aのアドレスに割り当てた場合、式x + 1は式&a[1]と同じことになります。同様に、式*(x + 1)は、値a[1]を参照します。演算子+=、+、または++によってポインタ値が増分されている箇所では、Dコンパイラにより、ポインタ演算が行われます。

ポインタ演算は、左側のポインタから整数が減算されている箇所、ポインタから別のポインタが減算されている箇所、ポインタに演算子 -- が適用されている箇所でも行われます。たとえば、次の D プログラムは、結果として 2 をトレースします。

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

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

汎用ポインタ

D プログラムでは、ポインタが参照するデータ型を指定せず、汎用ポインタアドレスを表現または操作すると便利な場合があります。汎用ポインタは、`void *` 型か、組み込みの型の別名 `uintptr_t` を使って指定します。キーワード `void` は、特定の型情報が存在しないことを示します。また、型の別名 `uintptr_t` は、現在のデータモデル内のポインタに適したサイズの符号なし整数型を示します。`void *` 型のオブジェクトにポインタ演算を適用することはできません。これらのポインタを間接参照するには、まず別の型にキャストする必要があります。ポインタ値に対して整数演算を行いたい場合は、ポインタを `uintptr_t` 型にキャストします。

`void` へのポインタは、連想配列の組の式や代入文の右式など、別のデータ型へのポインタが必要なとき、いつでも使用できます。同様に、`void` へのポインタが必要なときは、任意のデータ型のポインタを使用できます。`void` 以外のあるポインタ型の代わりに、`void` 以外の別の型のポインタを使用したい場合は、明示的キャストが必要になります。明示的キャストにより、ポインタを `uintptr_t` などの整数型に変換するか、これらの整数を適切なポインタ型に変換して戻す必要があります。

多次元配列

D では、多次元のスカラー配列は、あまり使用されません。多次元のスカラー配列は、ANSI-C との互換性を実現するためのものです。この機能を使って、C で作成されたオペレーティングシステムデータ構造を監視したり、使用したりできます。多次元配列は、基本型の後ろに、連続するスカラー配列サイズを角括弧 ([ ]) で囲んだ形式で宣言されます。たとえば、12 行 × 34 列の整数値から成る固定サイズの 2 次元四角形配列は、次のように宣言します。

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

多次元スカラー配列にも、同様の表記法でアクセスできます。たとえば、行 0 列 1 に格納されている値にアクセスしたい場合、次のような D 式を記述します。

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

多次元スカラー配列値の記憶域の位置は、「行番号」×「宣言された列数の合計」+「列番号」で計算されます。

多次元配列の構文と、連想配列アクセスの D 構文を混同しないでください。`a[0][1]` と `a[0, 1]` は、まったく別の構文です。連想配列に互換性のない組を使用したり、スカラー配列の連想配列アクセスを試行したりすると、D コンパイラからエラーメッセージが表示され、プログラムのコンパイルが拒否されます。

DTrace オブジェクトのポインタ

D コンパイラでは、連想配列、組み込み関数、変数などの DTrace オブジェクトへのポインタを、& 演算子を使って取得することは禁じられています。DTrace 実行環境では、プログラムに必要なメモリーをより効果的に管理するため、次のプローブが起動するまでの間に、必要に応じて変数のアドレスが再配置されます。このため、これらの変数のアドレスを取得することは禁じられています。複合構造を作成すれば、DTrace オブジェクト記憶域のカーネルアドレスを取得する式を作成することは可能です。このような式を D プログラム内で作成することはなるべく避けてください。こうした式を使用する必要がある場合は、プローブ起動時にアドレスをキャッシュしないようにしてください。

ANSI-C では、ポインタを使って、間接的な関数呼び出しや代入を実行できます。たとえば、代入演算子の左側に、単項の間接参照演算子 * を配置できます。D では、ポインタを使ったこのような式は許可されていません。値は、名前を指定するか、D スカラー配列または連想配列に配列インデックス演算子 [] を適用することにより、D 変数に直接代入しなければなりません。第 10 章「アクションとサブルーチン」に指定されているように、DTrace 環境に定義されている関数は名前を指定して呼び出す必要があります。D では、ポインタを使った間接的な関数呼び出しは許可されていません。

ポインタとアドレス空間

ポインタは、「仮想アドレス空間」内で物理メモリーへの翻訳を提供するアドレスです。DTrace は、オペレーティングシステムカーネル自体のアドレス空間内で D プログラムを実行します。Solaris システム全体で、多数のアドレス空間が管理されています。 オペレーティングシステムカーネル用のアドレス空間と、個々のユーザープロセス用のアドレス空間があります。各アドレス空間は、システム上のすべてのメモリーにアクセスできるように見えるので、複数のアドレス空間で同じ仮想アドレスポインタ値を再利用し、それぞれ異なる物理メモリーに翻訳できます。したがって、ポインタを含む D プログラムを作成するときは、使用するポインタに対応するアドレス空間を意識する必要があります。

たとえば、syscall プロバイダを使って、引数として整数ポインタまたは整数配列ポインタを取るシステムコール (例: pipe(2)) の開始を計測する場合、演算子 * や [] を使ってそのポインタまたは配列を間接参照することはできません。これは、そのアドレスが、システムコールを実行したユーザープロセスのアドレス空間内のアドレスだからです。D で、このアドレスに演算子 * または [] を適用すると、カーネルアドレス空間がアクセスされ、アドレスが無効であるというエラーメッセージが表示されます。または、このアドレスが有効なカーネルアドレスとたまたま同じだった場合、D プログラムに予想外のデータが返されます。

DTrace プローブからユーザープロセスメモリーにアクセスするには、第 10 章「アクションとサブルーチン」に記載されている copyin()、copyinstr()、copyinto() のうちいずれかの関数を、ユーザーアドレス空間ポインタに適用する必要があります。D プログラムの作成時には、混乱を防ぐため、ユーザーアドレスを適切に格納する変数を指定し、コメントを付けてください。ユーザーアドレスを間接参照するような D コードを誤ってコンパイルすることがないように、ユーザーアドレスを uintptr_t として格納することもできます。ユーザープロセスで DTrace を使用するテクニックについては、第 33 章「ユーザープロセスのトレース」を参照してください。

◆ ◆ ◆ 第 6 章

6 文字列

DTrace では、文字列をトレースし、操作できます。この章では、D 言語での文字列の宣言および操作の全機能について説明します。ANSI-C の場合とは異なり、D の文字列では、独自の組み込み型と演算子を使用できます。これらの型と演算子は、意味が明白で、トレースプログラム内で簡単に使用できます。

文字列表現

DTrace では、文字列は NULL バイト (値がゼロのバイトで、通常 `'\0'` と記述される) で終わる文字配列として表現されます。文字列の可視部分は可変長で、NULL バイトの位置によって長さが決まります。しかし、DTrace では、各プローブが決まった量のデータをトレースするように、各文字列が固定サイズの配列に格納されます。文字列の長さは、あらかじめ定義されたこの制限長内に収まっていなければなりませんが、この制限長は D プログラム内で変更できます。`dtrace` コマンド行で `strsize` オプションをチューニングして変更することもできます。チューニング可能な DTrace オプションについては、第 16 章「オプションとチューニング可能パラメータ」を参照してください。デフォルトの制限長は 256 バイトです。

D 言語で文字列を参照するときは、`char *` 型ではなく、明示的な `string` 型を使用します。`string` 型は、文字シーケンスのアドレスであるという点では `char *` 型と共通していますが、`string` 型の式では、D コンパイラや `trace()` などの D 関数の拡張機能を利用できます。たとえば、string 型では、文字列の実バイトをトレースするとき、`char *` 型のようなあいまいな点がありません。次の D 文を参照してください。

```
trace(s);
```

`s` が `char *` 型の場合、DTrace はポインタ `s` の値をトレースします。つまり、整数アドレス値がトレースされることになります。次の D 文を参照してください。

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

この場合、演算子 * の定義により、D コンパイラはポインタ s を間接参照し、その位置にある単一の文字をトレースします。この動作を利用して、単一の文字や、文字列でなく NULL バイトで終わっていないバイトサイズの整数配列を意図的に参照するような文字ポインタを操作できます。次の D 文を参照してください。

```
trace(s);
```

s が string 型の場合、この string 型は、変数 s にアドレスが格納されている NULL で終わる文字列をトレースするように、D コンパイラに指示を送ります。93 ページの「文字列比較」で説明するように、string 型の式の文字列大小比較を行うこともできます。

文字列定数

文字列定数は、二重引用符 (") で囲まれた形式で表され、D コンパイラにより自動的に string 型が割り当てられます。DTrace に割り当てられたシステムメモリーの容量が許す範囲で、任意の長さの文字列定数を定義できます。文字列定数を宣言すると、D コンパイラにより、自動的に終了 NULL バイト (\0) が付加されます。文字列定数オブジェクトのサイズは、この文字列のバイト数に終了 NULL バイト (1 バイト) を加えた長さです。

文字列定数には、改行文字をそのまま含めることはできません。改行文字を含めたい場合は、実際に改行文字を入力する代わりに、\n というエスケープシーケンスを入力します。文字列定数には、文字定数で使用できる特殊文字エスケープシーケンス (表 2–5 を参照) を含めることができます。

文字列代入

char * 変数の代入の場合とは異なり、文字列は、参照ではなく値ごとにコピーされます。文字列代入では、演算子 = により、元のオペランドの文字列の実バイトと NULL バイトが、左式の変数 (string 型) にコピーされます。string 型の変数を新しく作成するには、この変数に string 型の式を割り当てます。たとえば、次の D 文を参照してください。

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

この D 文では、string 型の新しい変数 s が作成され、そこに 6 バイトの文字列 "hello" (出力可能文字 5 バイトおよび NULL バイト) がコピーされます。文字列代入は、C ライブラリ関数 strcpy(3C) とよく似ていますが、元の文字列がコピー先の文字列の記憶容量を超過した場合、最終的な文字列では、超過分が自動的に切り捨てられます。

文字列変数に、文字列と互換性のある型の式を代入することもできます。この場合、元の式が D コンパイラにより自動的に string 型に拡張され、文字列代入が行われ

ます。D コンパイラには、char * 型または char[*n*] 型 (任意のサイズの char 型のスカラー配列) の任意の式を、string 型へと拡張する機能があります。

文字列変換

その他の型の式は、キャスト式の利用または特殊な演算子 stringof の適用により、明示的に string 型に変換できます。どちらの方法を使用しても同じことです。

```
s = (string) expression                s = stringof ( expression )
```

演算子 stringof は、右側のオペランドに非常に緊密に結合されます。通常、わかりやすいように式を丸括弧で囲みますが、必須ではありません。

ポインタ、整数、スカラー配列アドレスなど、スカラー型の式はすべて、文字列に変換できます。void など、その他の型の式は、string 型には変換できません。
誤って無効なアドレスを文字列に変換した場合でも、DTrace には保護機能があるので、システムや DTrace 自体に悪影響が及ぶことはありません。しかし、解読不能な文字シーケンスをトレースしてしまうことになります。

文字列比較

D では、二項関係演算子が多重定義されており、これらの演算子を使って、整数の比較だけでなく文字列の比較も行うことができます。関係演算子では、両方のオペランドが string 型であるか、一方のオペランドが string 型でもう一方のオペランドが string 型に拡張可能 (92 ページの「文字列代入」を参照) であれば、文字列比較が行われます。文字列比較には、すべての関係演算子を使用できます。

表 6–1　文字列で使用できる D の関係演算子

<	左のオペランドは右のオペランドより小さい
<=	左のオペランドは右のオペランド以下
>	左のオペランドは右のオペランドより大きい
>=	左のオペランドは右のオペランド以上
==	左のオペランドは右のオペランドと等しい
!=	左のオペランドは右のオペランドと等しくない

整数で使用する場合と同じく、各演算子は、int 型の値 (条件が真の場合は 1、偽の場合は 0) を返します。

関係演算子では、C ライブラリルーチン strcmp(3C) の場合と同様に、2 つの入力文字列がバイト単位で比較されます。個々のバイトの比較には、ASCII 文字セット内の対

応する整数値が使用されます (ascii(5) を参照)。NULL バイトが検出されるか、最大文字列長に達したら、比較は終了します。以下に、D の文字列比較とその結果の例を示します。

```
"coffee" < "espresso"          ... 1 (真) を返す
"coffee" == "coffee"           ... 1 (真) を返す
"coffee" >= "mocha"            ... 0 (偽) を返す
```

◆ ◆ ◆ 第 7 章

7 構造体と共用体

関連性のある変数同士は、「構造体」および「共用体」と呼ばれる複合データオブジェクトにグループ化することができます。D でこれらのオブジェクトを作成するには、新しい型定義を作成します。作成した新しい型は、連想配列の値を含むあらゆる D 変数で使用できます。この章では、こうした複合型を作成し操作するための構文とセマンティクスを紹介し、これらに使用する D 演算子について説明します。構造体と共用体の構文については、DTrace プロバイダ `fbt` と `pid` の使い方を示すプログラム例で説明します。

構造体

いくつかの型のグループから成る新しい型を作成するときは、D のキーワード `struct` (「*structure* (構造体)」の略) を使用します。この新しい構造体型を D の変数や配列の型として使用することにより、関連性のある複数の変数を単一の名前で定義できます。D の構造体は、C や C++ の対応する構造体と同じです。D の構造体は、Java プログラミングにおけるクラスに似ていますが、クラスとは違ってメソッドを持たず、データメンバーだけを備えています。

以下では、D を使って、シェルでシステムコール `read(2)` または `write(2)` が実行されるたびに、経過時間、呼び出し回数、引数として渡される最大バイト数などのデータを記録する、複雑なシステムコールトレースプログラムを作成してみましょう。次の例のように、3 つの連想配列を使ってプロパティを記録する D 節を記述できます。

```
syscall::read:entry, syscall::write:entry
/pid == 12345/
{
    ts[probefunc] = timestamp;
    calls[probefunc]++;
    maxbytes[probefunc] = arg2 > maxbytes[probefunc] ?
        arg2 : maxbytes[probefunc];
}
```

ただし、この節では、DTrace は 3 つの連想配列を作成し、それぞれに probefunc に対応する同一の組の値の各コピーを格納するため、効率面で問題があります。構造体を使用すると、読みやすく管理もしやすい、よりコンパクトなプログラムになります。まず、プログラムソースファイルの冒頭で新しい構造体型を宣言します。

```
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 */
};
```

キーワード struct の後ろに、この新しい型を参照する識別子 (オプション) を付けて、struct callinfo としています。続く中括弧 ({ }) 内には構造体のメンバーが記述され、宣言全体はセミコロン (;) で終わります。 構造体のメンバーの定義には、D 変数宣言と同じ構文を使用します。最初にメンバーの型を入力し、次に識別子名、最後にセミコロン (;) を入力します。

構造体の宣言には、新しい型を定義する働きしかありません。この宣言自体が、変数を作成したり、DTrace 内の記憶域を割り当てたりすることはありません。宣言後は、D プログラム内で struct callinfo を型として使用できます。構造体テンプレートを使用して、struct callinfo 型の変数 1 つにつき、4 つの変数のコピーが格納されます。メンバーは、メンバーリストの順序に従ってメモリー内に配置されます。データオブジェクトの配置に必要な場合は、メンバー間にパディングスペースが挿入されます。

個々のメンバーの値にアクセスするには、演算子「.」とメンバーの識別子名を使って、次のような式を作成します。

variable-name.*member-name*

以下に、新しい構造体型を使った改良版のプログラム例を示します。エディタで以下の D プログラムを入力し、rwinfo.d という名前のファイルに保存してください。

例 7–1 rwinfo.d: read(2) と write(2) の統計情報の収集

```
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 */
};

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

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

例 7–1 rwinfo.d: read(2) と write(2) の統計情報の収集 *(続き)*

```
{
    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);
}
```

プログラムを入力したら、シェルプロセスを 1 つ指定して、dtrace -q -s rwinfo.d を実行します。次に、シェルコマンドをいくつか入力します。シェルコマンドを入力し終わったら、dtrace 端末ウィンドウ内で Control-C キーを押します。すると、END プローブが起動し、結果が出力されます。

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

構造体のポインタ

C や D では、しばしば、ポインタを使って構造体を参照します。ポインタを使って構造体のメンバーにアクセスするときは、演算子 -> を使用します。メンバー m を持つ構造体 struct s があり、この構造体をポイントするポインタ sp がある場合 (sp は struct s * 型の変数)、メンバー m にアクセスする方法は 2 とおりあります。1 つは、演算子 * を使ってポインタ sp を間接参照する方法です。

```
struct s *sp;

(*sp).m
```

もう 1 つは、短縮形の演算子 -> を使用する方法です。sp が構造体のポインタである場合、次の 2 つの D コードの抜粋は、意味的に等しくなります。

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

DTrace では、構造体のポインタとして、curpsinfo、curlwpsinfo をはじめとするいくつかの組み込み変数を利用できます。curpsinfo と curlwpsinfo は、それぞれ構造体 psinfo と lwpsinfo を参照し、現在のプローブを起動したスレッドに関連付けられている現在のプロセスおよび軽量プロセス (LWP) の状態情報のスナップショットを提供します。Solaris LWP は、ユーザースレッドのカーネル表現です。このユーザースレッドの上に、Solaris スレッドと POSIX スレッドのインタフェースが構築されています。便宜上、DTrace は、この情報を /proc ファイルシステムの /proc/*pid*/psinfo ファイルや /proc/*pid*/lwps/*lwpid*/lwpsinfo ファイルと同じ形式でエクスポートします。/proc 構造体は、システムヘッダーファイル <sys/procfs.h> に定義されており、ps(1)、pgrep(1)、truss(1) などの監視/デバッグツールで使用されます。詳細は、proc(4) のマニュアルページを参照してください。以下に、例として、curpsinfo とその型および意味を用いた式を示します。

curpsinfo->pr_pid	pid_t	現在のプロセス ID
curpsinfo->pr_fname	char []	実行可能ファイル名
curpsinfo->pr_psargs	char []	最初のコマンド行引数

構造体定義の全容については、あとで <sys/procfs.h> ヘッダーファイルと proc(4) の該当箇所を参照して、確認してください。次の例では、pr_psargs メンバーを使ってコマンド行引数の照合を行うことにより、特定のプロセスを識別します。

構造体は、C プログラムで複雑なデータ構造を作成するときによく使用されます。同様に、D でも、構造体を記述し、参照することにより、Solaris オペレーティングシステムカーネルとそのシステムインタフェースの内部処理を効果的に監視できます。次の例では、ksyms(7D) ドライバと read(2) 要求の関係に注目して、先ほど紹介した構造体 curpsinfo といくつかのカーネル構造体について見ていきます。ドライバは、uio(9S) と iovec(9S) の 2 つの一般的な構造体を使って、文字デバイスファイル /dev/ksyms の読み取り要求に応答します。

構造体 uio にアクセスするときは、struct uio のように名前を指定するか、uio_t のように型の別名を指定します。この構造体は、カーネルとユーザープロセス間でのデータのコピーを伴う入出力要求を記述する際によく使用されるもので、詳細は uio(9S) のマニュアルページに記載されています。システムコール readv(2) または writev(2) を使って複数のチャンクが要求されると、そのたびに、入出力要求をそれぞれ部分的に記述した 1 つ以上の iovec(9S) 構造体から成る配列が、uio に格納され

ます。struct uio を操作するカーネルデバイスドライバインタフェース (DDI) ルーチンの中には、uiomove(9F) が含まれます。このルーチンは、カーネルドライバがユーザープロセスの read(2) 要求に応えてデータをユーザープロセスにコピーするために使用する、関数群の 1 つです。

ksyms ドライバは、文字デバイスファイル /dev/ksyms を管理しています。この文字デバイスファイルは、カーネルのシンボルテーブルに関する情報が収められた ELF ファイルのように見えますが、実際にはそうではありません。ドライバが、現在カーネルにロードされているモジュールセットを使用しているので、そのように見えるだけです。ドライバは、uiomove(9F) ルーチンを使って read(2) 要求に応答します。次の例では、/dev/ksyms から引数を指定して read(2) を呼び出す処理と、ドライバから uiomove(9F) を呼び出して read(2) に指定された位置のユーザーアドレス空間に結果をコピーする処理とが、同じであることを示します。

/dev/ksyms を強制的に読み取るには、strings(1) ユーティリティーに -a オプションを指定して実行します。シェルで strings -a /dev/ksyms を実行し、出力結果を確認してみましょう。エディタにスクリプト例の最初の節を入力し、ksyms.d という名前のファイルに保存してください。

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

この最初の節では、curpsinfo->pr_psargs という式を使って、strings(1) コマンドのコマンド行引数にアクセスし、これを照合しています。このため、スクリプトは、引数をトレースする前に正しい read(2) 要求を選択できます。char の配列になっている左側の引数と、文字列である右側の引数は、演算子 == で結ばれています。これにより、D コンパイラは、左側の引数を文字列に拡張し、文字列比較を行う必要があると判断します。シェルで dtrace -q -s ksyms.d コマンドを実行し、別のシェルで strings -a /dev/ksyms コマンドを実行してください。strings(1) を実行すると、DTrace から以下のような出力が得られます。

```
# 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
#
```

一般的な D のプログラミングテクニックを使ってこの例を拡張し、最初の read(2) 要求からカーネル内の下位スレッドへ下りていくようなプログラムも作成できます。syscall::read:entry でカーネルに入ると、次のスクリプトにより、このスレッドが対象のスレッドであることを示すスレッド固有のフラグ変数が設定されます。この

フラグは、syscall::read:return で消去されます。設定済みのフラグは、その他のプローブで、uiomove(9F) などのカーネル関数を計測する述語として使用できます。DTrace の関数境界トレース (fbt) プロバイダは、カーネル内に定義された関数 (DDI 内の関数を含む) の開始 (entry) プローブと終了 (return) プローブを発行します。次のソースコードは、fbt プロバイダを使って uiomove(9F) を計測するコードです。このコードを入力して、ksyms.d ファイルに保存してください。

例 7–2 ksyms.d: read(2) と uiomove(9F) の関係のトレース

```
/*
 * 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/
{
    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);
}
```

この例の最後の節では、スレッド固有変数 self->watched を使って、対象のカーネルスレッドが DDI ルーチン uiomove(9F) に入るタイミングを特定しています。そこから、組み込みの args 配列を使って、uiomove() の 4 番目の引数 (args[3]) にアクセスしています。この引数は、要求を表す struct uio のポインタになっています。D コンパイラは、args 配列の各メンバーに、計測されるカーネルルーチンの C 関数プロトタイプに当たる型を自動的に割り当てます。uio_iov メンバーには、要求の struct iovec のポインタが格納されます。このポインタのコピーは、節で使用するため、節固有変数 this->iov に保存されます。最後の文では、iovec のメンバーである

iov_len と iov_base にアクセスするため、this->iov を間接参照しています。iov_len と iov_base はそれぞれ、uiomove(9F) の長さ (バイト単位) と、最終的な基底アドレスを表しています。これらの値は、ドライバに対して発行されたシステムコール read(2) の入力パラメータと一致していなければなりません。シェルで dtrace -q -s ksyms.d コマンドを実行し、別のシェルで再度 strings -a /dev/ksyms コマンドを実行してください。次の例のような出力が得られます。

```
# 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
#
```

実際に出力されるアドレスとプロセス ID は、この例のとおりではありませんが、read(2) の入力引数は、ksyms ドライバから uiomove(9F) に渡されるパラメータと一致しています。

共用体

共用体も、ANSI-C と D の両方でサポートされる複合型です。共用体と構造体の間には、密接な関係があります。共用体では、異なった型を持つ複数のメンバーが定義されますが、メンバーオブジェクトはすべて同じ記憶域を占有します。つまり共用体は、時と場合に応じて有効なメンバーが変わる、バリアント型のオブジェクトであり、ある時点で有効なメンバーは、共用体の割り当て方によって決まります。通常、共用体のどのメンバーが有効になるかは、その他の変数や状態によって決定されます。共用体のサイズは、その最大のメンバーのサイズに一致しており、共用体のメモリー配置は、そのメンバーに必要な最大配置に一致しています。

Solaris の kstat フレームワークで定義されている構造体には、C と D の共用体について説明する以下の例で使用されている共用体が含まれます。kstat フレームワークは、メモリーの使用率や入出力スループットなどのカーネル統計情報を表す、名前付きのカウンタのセットをエクスポートするために使用されます。このフレームワークは、mpstat(1M) や iostat(1M) のようなユーティリティーの実装にも使用されます。このフレームワークは、struct kstat_named を使って、名前付きカウンタとその値を表します。このフレームワークの定義は以下のとおりです。

```
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 */
};
```

上記の宣言は、わかりやすくするために省略されています。構造体宣言の全容は、<sys/kstat.h> ヘッダーファイルで確認できます。また、kstat_named(9S) にも説明が記載されています。上記の宣言は、ANSI-C と D のどちらでも有効です。この宣言では、カウンタの型によって異なった型のメンバーを持つ共用体の値をメンバーとして含む、構造体が定義されています。共用体自体が別の型 struct kstat_named の内部に宣言されているので、共用体の正式名は省略されています。この宣言書式は、「無名共用体」として知られています。メンバー value は、先ほどの宣言で記述された共用体型ですが、この共用体型はほかの場所では使用されないので、名前を持っていません。構造体メンバー data_type には、struct kstat_named 型のオブジェクトごとに、どの共用体メンバーが有効になるかを示す値が割り当てられています。data_type の値には、C プリプロセッサトークンのセットが定義されています。たとえば、トークン KSTAT_DATA_CHAR はゼロであり、現在値がメンバー value.c に格納されていることを示しています。

例 7–3 は、ユーザープロセスをトレースすることにより、kstat_named.value という共用体にアクセスする様子を示しています。kstat_data_lookup(3KSTAT) 関数を使って、kstat カウンタの標本がユーザープロセスから収集され、この関数は struct kstat_named へのポインタを返します。mpstat(1M) ユーティリティーは、実行されるたびにこの関数を繰り返し呼び出し、最新のカウンタ値を収集します。シェルで mpstat 1 を実行し、出力結果を確認してください。しばらくしたら、Control-C キーを押して、mpstat を終了させます。収集したカウンタを確認するため、libkstat 内で mpstat コマンドが kstat_data_lookup(3KSTAT) 関数を呼び出すたびに起動するプローブを有効にします。このためには、新しい DTrace プロバイダ pid を使用します。pid プロバイダを使用すると、関数のエントリポイントなど、C のシンボル位置にあるユーザープロセス内で、動的にプローブを作成できます。次のようなプローブ記述を作成して、pid に、ユーザー関数の開始 (entry) 位置と終了 (return) 位置でプローブを作成させることもできます。

pid*process-ID*:*object-name*:*function-name*:entry

pid*process-ID*:*object-name*:*function-name*:return

たとえば、プロセス ID 内に、kstat_data_lookup(3KSTAT) の開始時に起動するプローブを作成したい場合、次のようなプローブ記述を作成します。

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

pid プロバイダは、プローブ記述に対応するプログラム位置で、指定したユーザープロセスに、動的計測機能を挿入します。このプローブ実装により、計測されるプログラム位置に到達したユーザースレッドは、オペレーティングシステムカーネルに割り込んで DTrace に入り、対応するプローブを起動するようになります。したがって、計測機能の場所がユーザープロセスに関連付けられていても、指定した DTrace の述語とアクションは、オペレーティングシステムカーネルのコンテキストで実行されます。pid プロバイダの詳細は、第 30 章「pid プロバイダ」に記載されています。

プログラムのコンパイル時に評価されて *dtrace* の追加コマンド行引数になる「マクロ変数」という識別子をプログラムに挿入すると、D プログラムを別のプロセスに適用するたびに D プログラムソースを編集する必要がなくなります。マクロ変数を指定するときは、ドル記号 ($) と、識別子として機能する数字を入力します。コマンド `dtrace -s` *script* `foo bar baz` を実行した場合、D コンパイラにより自動的に、マクロ変数 $1、$2、$3 がそれぞれトークン foo、bar、baz として定義されます。マクロ変数は、D プログラム式やプローブ記述で使用します。たとえば、次のプローブ記述では、dtrace の追加引数として指定されているプロセス ID を計測できます。

```
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;
}
```

マクロ変数と再利用可能なスクリプトの詳細は、第 15 章「スクリプトの作成」で説明します。プロセス ID を使ってユーザープロセスを計測する方法がわかったところで、共用体の標本の収集を再開しましょう。次のソースコード例をエディタに入力し、kstat.d という名前で保存してください。

例 7–3　kstat.d: kstat_data_lookup(3KSTAT) への呼び出しのトレース

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

例 7–3 kstat.d:kstat_data_lookup(3KSTAT) への呼び出しのトレース (続き)

```
}

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;
}
```

シェルで mpstat 1 コマンドを実行して、統計情報の標本を収集し、毎秒報告するモードで mpstat(1M) の実行を開始します。mpstat の実行開始後、別のシェルでコマンド dtrace -q -s kstat.d ‘pgrep mpstat‘ を実行します。アクセス対象の統計情報に対応した出力が得られます。dtrace を強制終了し、シェルプロンプトに戻るには、Control-C キーを押します。

```
# 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
#
```

各端末ウィンドウ内の出力を取得し、統計情報の値をそれぞれ 1 つ前の値から次々に減算していけば、dtrace の出力と mpstat の出力の相関関係が明らかになりま

す。このプログラム例は、ルックアップ関数に入るときにカウンタ名のポインタを記録し、kstat_data_lookup(3KSTAT) が終了するときにほとんどのトレース作業を実行します。D の組み込み関数 copyinstr() と copyin() は、戻り値 arg1 が NULL 以外のとき、ユーザープロセスから DTrace に関数の実行結果をコピーします。kstat データのコピーが完了すると、共用体の ui64 カウンタ値が報告されます。この単純な例では、mpstat が value.ui64 メンバーを使用するカウンタの標本を収集すると想定しています。練習として、複数の述語を使用し、data_type メンバーに対応する共用体メンバーを出力するよう、kstat.d を書き直してみてください。また、連続するデータ値の差分を計算して mpstat に似た結果を実際に出力するような kstat.d も作成してみてください。

メンバーのサイズとオフセット

演算子 sizeof を使用すると、構造体や共用体を含むあらゆる D 型 (D 式) のサイズ (バイト単位) を特定できます。演算子 sizeof は、式または丸括弧で囲まれた形式の型名に適用できます。次の 2 つの例を参照してください。

```
sizeof expression
sizeof (type-name)
```

たとえば、式 sizeof (uint64_t) の戻り値は 8 です。式 sizeof (callinfo.ts) も、先ほどのプログラム例のソースコードに挿入した場合、8 を返します。演算子 sizeof の正式な戻り型は、size_t です。これは、バイト数を表現する際に使用される型の別名であり、定義により、現在のデータモデル内のポインタと同じサイズの符号なし整数になります。式に演算子 sizeof を適用した場合、この式は D コンパイラによって評価されますが、結果のオブジェクトサイズはコンパイル時に計算され、式のコードは生成されません。sizeof は、整数定数が必要な箇所で使用できます。

同様の演算子 offsetof を使って、構造体型または共用体型のオブジェクトに関連付けられた記憶域の開始位置から、構造体メンバーまたは共用体メンバーのオフセットをバイト単位で特定することもできます。演算子 offsetof は、次の形式の式で使用します。

```
offsetof (type-name, member-name)
```

type-name は構造体型の名前、共用体型の名前、またはこれらの型の別名で、*member-name* はその構造体または共用体のメンバーの名前を示す識別子です。offsetof は、sizeof と同様に、size_t を返し、D プログラム内の整数定数を使用できる箇所で使用できます。

ビットフィールド

D では、任意のビット数の整数の構造体メンバーまたは共用体メンバーも定義できます。これを「ビットフィールド」と呼びます。ビットフィールドを宣言するときは、次の例のように、符号付きまたは符号なしの整数基本型、メンバー名、およびフィールドに割り当てるビット数を示す接尾辞を指定します。

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

ビットフィールド幅は、メンバー名の後ろのコロンに続く整数定数で表されます。ビットフィールド幅は必ず正の数で、対応する整数基本型の幅を超えない範囲のビット数とします。D では、64 ビットを超えるビットフィールドは宣言できません。D のビットフィールドは、対応する ANSI-C の機能と互換性があり、ANSI-C の機能にアクセスできます。通常、ビットフィールドは、メモリー記憶域が不足している場合や、構造体レイアウトをハードウェアレジスタレイアウトと一致させる必要がある場合に使用します。

ビットフィールドは、整数やマスクセットのレイアウトを自動化し、メンバー値を抽出するコンパイラ構成です。マスクをユーザー定義し、演算子 & を使用しても、同じ結果が得られます。C と D のコンパイラは、できるだけ効率よくビットを詰めようとしますが、処理の順序や方法は特に指定されていません。そのため、使用するコンパイラやアーキテクチャが異なる場合は、ビットフィールドを指定しても、同一のビットレイアウトが得られるとはかぎりません。安定したビットレイアウトが必要な場合は、ビットマスクを作成し、演算子 & を使って値を抽出してください。

ビットフィールドメンバーにアクセスするときは、構造体や共用体のメンバーにアクセスするときと同じく、その名前と演算子「.」または「->」を指定します。ビットフィールドは、どの式でも使用できるようにするため、自動的に 2 番目に大きい整数型に拡張されます。ビットフィールド記憶域はバイト境界上には配置できません。また、そのサイズは、バイト数の概数であってはなりません。したがって、ビットフィールドメンバーには、演算子 `sizeof` や `offsetof` を適用することはできません。D コンパイラでは、演算子 & を使ってビットフィールドメンバーのアドレスを指定することも禁じられています。

◆ ◆ ◆ 第 8 章

8 型と定数の定義

この章では、D で型の別名と名前付き定数を宣言する方法について説明します。D プログラムでの型と名前空間の管理、オペレーティングシステムの型と識別子についても説明します。

Typedef

キーワード `typedef` は、既存の型の別名として識別子を宣言するときに使用します。D 型の宣言がすべてそうであるように、キーワード `typedef` は、以下の形式の宣言で、プローブ節の外側に記述します。

```
typedef existing-type new-type ;
```

existing-type は任意の型の宣言、*new-type* はこの型の別名として使用する識別子です。たとえば次のような宣言があるとします。

```
typedef unsigned char uint8_t;
```

この宣言が D コンパイラで内部使用されることにより、`uint8_t` という型の別名が生成されます。型の別名は、標準の型と同じように使用できます。たとえば、変数の型、連想配列値の型、組のメンバーの型として使用できます。`typedef` は、新しい `struct` の宣言など、さらに複雑な宣言と組み合わせることができます。

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

この例の `struct foo` には、その別名 `foo_t` と同じ型が定義されています。Solaris の C システムヘッダーでは、`typedef` の別名は、接尾辞 `_t` で表されることがよくあります。

列挙

プログラム内の定数の記号名を定義すると、読みやすく、将来的に管理しやすいプログラムになります。このためには、「列挙」を定義します。列挙は、一連の整数と、「列挙子」と呼ばれる一連の識別子を関連付けます。コンパイラは、この列挙子を認識し、その位置に対応する整数値を代入します。列挙の定義には、次のような宣言を使用します。

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

この列挙の最初の列挙子 RED には値ゼロ、以降の列挙子には、前の列挙子に割り当てられた値の次の整数値が割り当てられます。各列挙子に明示的に整数値を指定することもできます。この場合は、次の例のように、列挙子と整数定数を等号で結びます。

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

列挙子 BLUE には値が指定されていませんが、前の列挙子が 9 に設定されているので、コンパイラによって値 10 が割り当てられます。列挙の定義後は、D プログラム内のどこでも整数定数と同じように列挙子を使用できます。列挙 enum colors は、int と同等の型としても定義されています。enum 型の変数は、int 型と同じように使用できます。また、enum 型の変数には、任意の整数値を割り当てることができます。また、型名が不要なときは、宣言内で enum 名を省略できます。

列挙子は、プログラム内の以降のすべての節、すべての宣言で使用できます。そのため、複数の列挙で同じ列挙子を定義することはできません。ただし、1 つの列挙、または複数の列挙内に、同じ値を持つ列挙子を複数個定義することは問題ありません。列挙型の変数に、対応する列挙子を持たない整数を割り当てることもできます。

D の列挙の構文は、ANSI-C の場合と同じです。D でも、オペレーティングシステムカーネルやそのロード可能なモジュールに定義された列挙にアクセスできます。ただし、これらの列挙の列挙子は、D プログラム全体で使用可能であるわけではありません。カーネル列挙子が使用可能であるのは、二項比較演算子を使って対応する列挙型のオブジェクトとの比較を行う際、その比較演算子の引数として指定された場合にかぎられます。たとえば、関数 uiomove(9F) は、次のように定義された enum uio_rw 型のパラメータを持ちます。

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

列挙子 UIO_READ と UIO_WRITE は、通常、D プログラム内で使用可能ではありません。enum uio_rw 型の値の比較を行うことによって列挙子を拡張すれば、使用可能になります。次の例を参照してください。

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

この例では、enum uio_rw 型の変数 args[2] と列挙子 UIO_WRITE の比較によって、書き込み要求の uiomove(9F) 関数呼び出しがトレースされます。左側の引数は列挙型なので、D コンパイラは、右側の識別子を解決する際、列挙を検索します。オペレーティングシステムカーネルには大量の列挙が定義されていますが、この機能を利用すれば、D プログラム内で重複した識別子名を誤って使用してしまうのを防ぐことができます。

インライン

D の名前付き定数は、inline 指令を使って定義することもできます。この指令を使用すると、より一般的な方法で、コンパイル時に事前に定義された値や式で置き換えられる識別子を作成できます。インライン指令は、C プリプロセッサで使用される #define 指令よりも効果的な文字列置換形式です。これは、インライン指令では置換に実際の型が割り当てられ、単なる構文素ではなくコンパイル済み構文ツリーを使って置換できるからです。インライン指令は、次の形式の宣言を使って指定します。

```
inline type name = expression ;
```

type は既存の型の型定義、*name* は以前にインライン変数または大域変数として定義されていない任意の有効な D 識別子、*expression* は任意の有効な D 式です。インライン指令の処理が完了すると、D コンパイラにより、プログラムソース内の後続の各 *name* インスタンスが、コンパイル済みの *expression* で置き換えられます。たとえば、次の D プログラムでは、文字列 "hello" と整数値 123 がトレースされます。

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

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

インライン名は、対応する型の大域変数と同じように使用できます。コンパイル時に、インライン式が整数定数または文字列定数に入れられた場合、スカラー配列サイズなど、定数式を必要とする状況でもインライン名を使用できます。

インライン指令の評価時には、インライン式の構文エラーのチェックも行われます。式の結果の型は、D 代入演算子 (=) と同じ規則に従って、インラインで定義された型と互換性のある型でなければなりません。インライン式でインライン識別子そのものを参照することはできません。 再帰的定義は禁じられています。

DTrace ソフトウェアパッケージは、システムディレクトリ `/usr/lib/dtrace` に多数の D ソースファイルをインストールします。このディレクトリには、D プログラム内で使用できるインライン指令が格納されています。たとえば、`signal.d` ライブラリには、次のような指令が格納されています。

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

これらのインライン定義を使用すると、`signal(3HEAD)` のマニュアルページに記載されている最新の Solaris のシグナル名群にアクセスできます。同様に、`errno.d` ライブラリには、`Intro(2)` のマニュアルページに記載されている C の `errno` 定数のインライン指令が格納されています。

デフォルトでは、D コンパイラには、提供されているすべての D ライブラリファイルが自動的に格納されます。そのため、これらの定義はすべての D プログラムで使用できます。

型の名前空間

ここでは、D の名前空間と型関連の問題について説明します。ANSI-C をはじめとする従来の言語では、型の可視性は、その型が関数やその他の宣言内に入れ子になっているかどうかによって決定されます。C プログラムの外側 (外部スコープ) で宣言された型には、単一の大域名前空間が関連付けられます。このため、これらの型は、プログラム内のどの位置でも可視的に使用できます。外部スコープの代表例として、C ヘッダーファイルに定義された型が挙げられます。D では、従来の言語とは異なり、複数の外部スコープの型にアクセスできます。

D 言語は、オペレーティングシステムカーネル、これに関連付けられているロード可能なカーネルモジュールのセット、システム上で実行されるユーザープロセスなど、ソフトウェアスタックの複数の層を動的に監視できるように設計されています。単一の D プログラムでプローブをインスタンス化し、複数のカーネルモジュールのデータや、独立したバイナリオブジェクトにコンパイルされるその他のソフトウェア構成要素のデータを収集できます。このため、DTrace や D コンパイラに、使用可能な型として、同じ名前の複数のデータ型 (通常、定義はそれぞれ異なる) が提

供されることがあります。この問題に対処するため、D コンパイラでは、各型に、それを含むプログラムオブジェクト別の名前空間が割り当てられます。特定のプログラムオブジェクトの型にアクセスするには、型名に、オブジェクト名と逆引用符 (`) で表されるスコープ演算子を指定します。

たとえば、カーネルモジュール `foo` に次の C 型宣言が含まれているとします。

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

この場合、D から `struct bar` 型や `bar_t` 型にアクセスするには、次のように型名を使用します。

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

逆引用符の演算子は、型名が適切であれば、どのような状況でも使用できます。たとえば、D 変数宣言の型を指定するときや、D プローブ節にキャスト式を指定するときに使用できます。

D コンパイラでは、それぞれ名前 `C` と `D` を使用する、2 つの特殊な組み込み型の名前空間を使用できます。C の型の名前空間は、最初に、`int` のような標準 ANSI-C 組み込み型に割り当てられています。さらに、C プリプロセッサ cpp(1) によって実行された `dtrace -C` オプションの結果得られた型定義が処理され、C スコープに追加されます。その結果、コンパイルエラーを発生させずに、別の型の名前空間で使用可能な型宣言を含む C ヘッダーファイルを追加できます。

D の型の名前空間は、最初に、`int`、`string` をはじめとする D の組み込み型と、`uint32_t` などの D の組み込み型の別名に割り当てられています。D プログラムソース内の型宣言は、新しく宣言されるたびに、自動的に D の型の名前空間に追加されます。D プログラム内で、ほかの名前空間のメンバーの型から成る複合型 (`struct` など) を作成した場合、宣言により、そのメンバーの型が D の名前空間にコピーされます。

D コンパイラは、名前空間が明示的に指定されていない (逆引用符の演算子が使用されていない) 型宣言を検出すると、有効な型の名前空間を検索し、指定された型名と一致する型名を見つけようとします。常に C の名前空間が最初に検索され、そのあとで D の名前空間が検索されます。C と D のどちらの名前空間でも型名が見つからない場合、有効なカーネルモジュールの型の名前空間が、カーネルモジュール ID の昇順で検索されます。この順番では必ず、コアカーネルを構成するバイナリオブジェクトがロード可能なカーネルモジュールより先に検索されます。ただし、ロード可能なモジュール間の検索順序は指定されていません。ロード可能なカーネルモジュールに定義された型にアクセスするときは、ほかのカーネルモジュールとの型名の競合を避けるため、スコープ演算子を使用してください。

D コンパイラは、C のインクルードファイルにアクセスすることなくオペレーティングシステムのソースコードに関連付けられた型に自動的にアクセスするため、Solaris

のコアカーネルモジュールから提供される圧縮形式の ANSI-C デバッグ情報を使用します。このシンボリックなデバッグ情報は、システム上のすべてのカーネルモジュールに提供されるわけではありません。ユーザーがアクセスしようとした型の名前空間のモジュールに、DTrace で使用される圧縮形式の C デバッグ情報が含まれていない場合、D コンパイラからエラーが返されます。

◆ ◆ ◆ 第 9 章

集積体

システムパフォーマンスについて計測するときは、個々のプローブによって収集されたデータについて考えるよりも、データをどのように集積できるかについて考えたほうが、明確な答えを得やすくなります。たとえば、ユーザー ID ごとにシステムコールの回数を調べたい場合、*1* 回 *1* 回のシステムコールで収集されたデータについて考慮する必要はありません。ユーザー ID とシステムコールの表を確認できれば、それで十分です。従来、このような調査が必要な場合は、システムコールごとにデータを収集し、`awk(1)` や `perl(1)` などのツールを使って、収集したデータに後処理を行なっていました。しかし、DTrace では、データの集積こそがもっとも重要な操作になります。この章では、DTrace の「集積体」の処理機能について説明します。

集積関数

次のようなプロパティを持つ関数を「集積関数」と呼びます。

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

x_n は、任意のデータセットです。つまり、データのサブセットに集積関数を適用し、その結果に再度集積関数を適用すると、データ全体に集積関数を適用した場合と同じ結果になります。たとえば、指定のデータセットの合計を計算する関数、`SUM` について考えてみましょう。生のデータが {2, 1, 2, 5, 4, 3, 6, 4, 2} である場合、このデータセット全体に `SUM` を適用した結果は {29} になります。同様に、最初の 3 つの要素で構成されたサブセットに `SUM` を適用した結果は {5}、続く 3 つの要素に `SUM` を適用した結果は {12}、残りの 3 つの要素に `SUM` を適用した結果は {12} になります。これらの結果である {5, 12, 12} に `SUM` を適用すると、元のデータに `SUM` を適用したときと同じ結果 (ここでは {29}) が得られます。このため、SUM は「集積関数」と呼ばれます。

すべての関数が集積関数として機能するわけではありません。たとえば、データセットの中央値を計算する関数、`MEDIAN` は、集積関数ではありません。中央値とは、データセット内の要素を大きさ順に並べたとき、ちょうど真ん中にくる要素のことです。`MEDIAN` を得るには、データセットをソートして、その真ん中の要素を選

択します。最初の生のデータに戻りましょう。最初の 3 つの要素から成るデータセットに MEDIAN を適用した場合、結果は {2} になります (ソート済みセットは {1, 2, 2} で、中央の要素は {2} です)。同様に、次の 3 つの要素に MEDIAN を適用した結果は {4}、最後の 3 つの要素に MEDIAN を適用した結果は {4} になります。このように、3 つの要素から成る各サブセットに MEDIAN を適用した場合、{2, 4, 4} というデータセットが得られます。このデータセットに MEDIAN を適用すると、結果は {4} になります。これに対して、元のデータセットをソートすると、{1, 2, 2, 2, 3, 4, 4, 5, 6} になります。このデータセットに MEDIAN を適用すると、結果は {3} になります。このように、結果が一致しないので、MEDIAN は集積関数ではありません。

データセット関連の情報を得たいときに使用する関数の多くは、集積関数です。これらの関数を使って、データセット内の要素数を数えたり、データセットの最小値や最大値を求めたり、データセット内のすべての要素の合計を計算したりできます。データセットの平均値を特定するには、セット内の要素数を数える関数と、セット内の要素を合計する関数を使用します。

ただし、集積関数以外にも、便利な関数は存在します。たとえば、データセットのモード (もっともよく使用する要素) を計算する関数、データセットの中央値を計算する関数、データセットの標準偏差を計算する関数などです。

データのトレース時に集積関数を適用すると、多数の利点があります。

- データセット全体を格納する必要がありません。セットに新しい要素が追加されるたびに、現在の中間結果と新しい要素から成るセットが作成され、集積関数が計算されます。新しい結果の計算が完了したら、新しい要素は破棄してかまいません。この処理により、データポイント数が非常に多くなっても、記憶域の消費量を少なく抑えることができます。
- データ収集が原因で、スケーラビリティに問題が起こることはありません。集積関数は、中間結果を共用のデータ構造に格納せず、*CPU* 単位で保存します。その後、この CPU 単位の中間結果のセットに集積関数が適用され、システム全体の最終的な結果が得られます。

集積体

DTrace は、集積関数の実行結果を「集積体」と呼ばれるオブジェクト内に格納します。この集積体の結果には、連想配列で使用するものとよく似た式の組でインデックスが付けられます。D の集積体の構文は、次のとおりです。

```
@name[ keys ] = aggfunc ( args );
```

name は集積体の名前、*keys* は複数の D 式をコンマで区切った形式のリスト、*aggfunc* は DTrace 集積関数、*args* は指定された集積関数の引数をコンマで区切った形式のリストです。集積体 *name* は、特殊文字 @ で始まる D 識別子です。D プログラム内で使用するすべての集積体は大域変数です。スレッド固有の集積体、節固有の集積体は

存在しません。集積体名は、その他の D 大域変数とは別の識別子用名前空間に格納されます。名前を再利用する場合、a と @a はまったく別の変数になります。単純な D プログラム内で、名前のない集積体を指定するときは、特殊な集積体名 @ を使用します。D コンパイラは、この名前を集積体名 @_ の別名と解釈します。

以下の表に、DTrace の集積関数を示します。ほとんどの集積関数は、新しいデータを表す引数を 1 つとるだけです。

表 9–1　DTrace の集積関数

関数名	引数	結果
count	なし	呼び出された回数。
sum	スカラー式	指定された式の合計値。
avg	スカラー式	指定された式の算術平均。
min	スカラー式	指定された式のうちもっとも小さい値。
max	スカラー式	指定された式のうちもっとも大きい値。
lquantize	スカラー式、下限値、上限値、ステップ値	指定された式の値から成る、指定された範囲の線形度数分布。指定された式より小さい、最大バケット内の値を増分します。
quantize	スカラー式	指定された式の値の二乗分布。指定された式より小さい、2 のべき乗の最大バケット内の値を増分します。

たとえば、システム内の write(2) システムコールの回数をカウントしたい場合、通知文字列をキーに指定して、集積関数 count() を使用します。

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

デフォルトの設定では、dtrace コマンドを実行すると、プロセスの終了時に、それが明示的な END アクションの結果であるかユーザーが Control-C キーを押したためかにかかわらず、集積体の結果が出力されます。このコマンドを実行し、しばらく待ってから Control-C キーを押すと、次のような結果が得られます。

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

  write system calls                                              179
#
```

プロセス名ごとのシステムコール数をカウントしたい場合は、集積体のキーとして execname 変数を指定します。

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

このコマンドを実行し、しばらく待ってから Control-C キーを押すと、次のような結果が得られます。

```
# 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
#
```

書き込みの情報として、実行可能ファイルの名前とファイル記述子の両方を出力することもできます。ファイル記述子は、write(2) の最初の引数です。次の例では、キーとして execname と arg0 の両方が指定されています。

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

このコマンドを実行すると、実行可能ファイルの名前とファイル記述子を含む、次の例のような表が出力されます。

```
# 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
#
```

次の例では、write システムコールにかかった時間の平均が、プロセス名ごとに出力されます。この例では、平均を求める式を引数にとる集積関数 `avg()` が使用されています。したがって、システムコールにかかった時計時間の平均が求められます。

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

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

このコマンドを実行し、しばらく待ってから Control-C キーを押すと、次のような結果が得られます。

```
# 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
#
```

平均値はさまざまな場面で利用できる情報ですが、通常、この情報だけでは、データポイントの分布まではわかりません。データ分布の詳細を確認したい場合は、次の例のように、集積関数 quantize() を使用します。

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

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

このスクリプトの出力は、先ほどの例の出力よりも長くなります。これは、先ほどの例で 1 行に出力されていた内容が、度数分布表として出力されるからです。以下は、出力例の抜粋です。

```
  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
```

度数分布の行は必ず 2 のべき乗の値になります。各行の右側に表示されているカウントは、その行の左側に表示されている値以上、次の行の左側に表示されている値未満に対応する要素数です。たとえば、上の出力例からは、`iropt` が、8,192 ナノ秒から 16,383 ナノ秒の間に 4,149 回の書き込みを行なっていることがわかります。

`quantize()` は、データの内容を短時間で把握するには便利ですが、線形値の分布を調べたい場合もあります。線形値の分布を表示するには、集積関数 `lquantize()` を使用します。`lquantize()` 関数を使用するときは、D 式と 3 つの引数 (下限値、上限値、ステップ値) を指定します。たとえば、ファイル記述子別に書き込みの分布を調べたい場合、2 のべき乗の量子化では効率がよくありません。そこで、次のように、狭い範囲で線形量子化を行います。

```
syscall::write:entry
{
    @fds[execname] = lquantize(arg0, 0, 100, 1);
}
```

このスクリプトは、数秒実行しただけで大量の情報を出力します。以下は、一般的な出力例の抜粋です。

```
  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
```

集積関数 `lquantize()` では、過去のある時点からのデータを集積することもできます。このテクニックを使って、時間の経過とともに変化する動作を監視できます。以下の例では、`date(1)` コマンドを実行中のプロセスにおける、システムコールの動作の変化を確認できます。

```
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;
}
```

このスクリプトでは、複数の date(1) プロセスを実行しているときのシステムコールの動作を詳しく調べることができます。結果を表示するには、D スクリプトの実行中に別のウィンドウで `sh -c 'while true; do date >/dev/null; done'` を実行します。すると、date(1) コマンドのシステムコールの動作の概要が出力されます。

```
# 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
```

この出力から、カーネルを必要とするサービスについて、date(1) コマンドのさまざまなフェーズの概要を確認できます。これらのフェーズの詳細を把握するには、いつ、どのシステムコールが呼び出されたかを確認します。この場合は、定数文字列ではなく変数 probefunc について集積するように、D スクリプトを変更します。

集積体の出力

デフォルトの設定では、複数の集積体が、D プログラムに記述された順番で表示されます。この設定を変更するには、集積体を出力する printa() 関数を使用します。第 12 章「出力書式」でも説明しますが、printa() 関数は、集積体データの書式を書式文字列を使って正確に設定したい場合にも使用します。

D プログラム内の printa() 文で集積体の書式が設定されていない場合、dtrace コマンドを実行すると、集積体データのスナップショットが生成され、その結果はトレースの完了後に 1 回だけ、デフォルトの集積体の書式で出力されます。printa() 文で集積体の書式が設定されている場合、デフォルトの動作は無効になります。プログラムの dtrace:::END プローブ節に printa(@*aggregation-name*) という文を追加しても、同じ結果が得られます。集積関数 avg()、count()、min()、max()、および sum() のデフォルトの出力書式では、各組の集積値に対応する 10 進整数値が出力されます。集積関数 lquantize() と quantize() のデフォルトの出力形式では、結果の ASCII テーブルが出力されます。集積体の組の出力は、個々の組要素に trace() を適用した場合と同じになります。

データの正規化

一定期間のデータを集積する際、定数係数を使用してデータを正規化できます。このテクニックを使用すると、互いに素のデータを簡単に比較できます。たとえば、システムコールを集積するとき、システムコールを経過時間の絶対値としてではなく、秒当たりのレートで出力できます。DTrace の normalize() アクションでは、この方法でデータを正規化できます。normalize() のパラメータとしては、集積体と正規化係数を指定します。集積体の結果としては、個々の値を正規化係数で割った値が出力されます。

以下の例では、データをシステムコールごとに集積する方法を示します。

```
#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);
}
```

このスクリプトをしばらく実行すると、デスクトップマシンに次のような結果が出力されます。

```
# 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() は、指定された集積体に正規化係数を設定しますが、このアクションによって配下のデータが変更されることはありません。denormalize() は、集積体だけをとります。上記の例に非正規化アクションを追加すると、生のシステムコールカウントと秒当たりのレートの両方が出力されます。

```
#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);
}
```

上のスクリプトをしばらく実行すると、次の例のような出力が得られます。

```
# 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
```

集積体は、再正規化も可能です。同じ集積体に対して複数回 `normalize()` を呼び出した場合、直前の呼び出しで指定された係数が正規化係数になります。以下の例では、時間の経過とともに秒当たりのレートが出力されます。

例 9–1 renormalize.d: 集積体の再正規化

```
#pragma D option quiet

BEGIN
{
    start = timestamp;
}

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

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

集積体の消去

DTrace で単純な監視スクリプトを作成しているとき、clear() 関数を使って、集積体内の値を定期的に消去できます。この関数には、パラメータを 1 つだけ指定できます。パラメータとして指定できるのは、集積体だけです。clear() 関数で消去されるのは、集積体の値だけです。集積体のキーは保持されます。したがって、集積体内に値ゼロのキーがある場合は、「このキーの値はかつてはゼロではなかったが、*clear* によってゼロが設定された」と解釈できます。()集積体の値とキーの両方を破棄するときは、trunc() を使用します。詳細については、127 ページの「集積体の切り捨て」を参照してください。

以下は、例 9–1 に clear() を追加した例です。

```
#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;
}
```

例 9–1 では dtrace の呼び出し開始から終了までのシステムコールレートが出力されましたが、上記の例では、過去 10 秒間のシステムコールレートだけが出力されます。

集積体の切り捨て

集積体の結果を調べるときは、通常、上位結果の数行だけに注目します。上位の値以外に関連付けられたキーや値は、あまり重要ではありません。また、キーと値の両方を削除して、集積体の結果全体を破棄したい場合もあります。DTrace の trunc() 関数は、こうした状況で使用します。

trunc() のパラメータとして指定できるのは、集積体と切り捨て値 (オプション) です。trunc() で切り捨て値を省略すると、集積体全体で集積体値と集積体キーとの両方が破棄されます。trunc() で切り捨て値 *n* を指定した場合は、上位 *n* 個の値に関連付けられている集積体値と集積体キー以外の値とキーを破棄します。つまり、trunc(@foo) では集積体全体が破棄されるのに対し、trunc(@foo, 10) では、foo という名前の集積体値の上位 10 個が保持され、それ以外の値が切り捨てられることになります。切り捨て値として 0 を指定した場合も、集積体全体が破棄されます。

上位 *n* 個の値ではなく下位 *n* 個の値を参照したい場合は、trunc() に負の切り捨て値を指定します。たとえば、trunc(@foo, -10) では、foo という名前の集積体値の下位 10 個が保持され、それ以外の値が切り捨てられます。

以下の例は、10 秒間での上位 10 個のシステム呼び出しアプリケーションについて秒当たりのシステムコールレートが出力だけされるように、先ほどのシステムコールの例を拡張したものです。

```
#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;
}
```

以下は、負荷の少ないラップトップでこのスクリプトを実行したときの出力例です。

```
  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
```

欠落の最小化

DTrace は、カーネルに集積体データの一部を格納します。このため、集積体に新しいキーを追加したとき、容量が不足することがあります。この場合、カウンタの値は大きくなりますが、データは落とされ、`dtrace` から集積体欠落を示すメッセージが表示されます。DTrace は、容量が動的に増加する場合のユーザーレベルで長時間実行状態 (集積体キーと中間結果で構成される) を保存します。このため、集積体欠落はめったに発生しません。それでも集積体欠落が発生しそうなときは、`aggsize` オプションを指定して集積体バッファーサイズを大きくすることにより、欠落が発生する確率を低減できます。このオプションは、DTrace のメモリーフットプリントを最小化するときにも使用します。`aggsize` は、ほかのサイズオプションと同じく、任

意のサイズの接尾辞とともに指定できます。投機バッファーのサイズ変更ポリシーは、bufresize オプションで指定します。バッファリングの詳細については、第 11 章「バッファーとバッファリング」を参照してください。オプションの詳細については、第 16 章「オプションとチューニング可能パラメータ」を参照してください。

ユーザーレベルで集積体データが消費されるレートを増やすことによっても、集積体欠落の発生を抑えることができます。デフォルトでは、このレートは毎秒 1 回ですが、この値は、aggrate オプションを指定して明示的にチューニングできます。aggrate は、その他のレートオプションと同じく、任意の時間接尾辞とともに指定できます。ただし、デフォルトでは、秒当たりのレートが設定されています。aggsize オプションの詳細については、第 16 章「オプションとチューニング可能パラメータ」を参照してください。

◆ ◆ ◆ 第 10 章

10 アクションとサブルーチン

trace() や printf() といった D 関数を呼び出すことで、DTrace の 2 種類のサービスを呼び出すことができます。 サービスのうち 1 つは、データをトレースしたり DTrace 外部の状態を変更したりする「アクション」です。もう 1つは、DTrace 内部の状態にのみ影響を及ぼす「サブルーチン」です。この章では、これらのアクションとサブルーチンを定義し、その構文と意味について説明します。

アクション

アクションには、DTrace プログラムと DTrace 外部のシステムの相互作用を促す働きがあります。もっとも一般的なアクションは、DTrace バッファーへデータを記録するアクションです。そのほかにも、現在のプロセスを停止するアクション、現在のプロセス上で特定のシグナルを発行するアクション、トレースをすべて停止するアクションなどがあります。こうしたアクションのうち、明確な方法でシステムに変更を加えるものを「破壊アクション」と呼びます。破壊アクションは、明示的に有効化されている場合にだけ使用できます。データの記録アクションは、デフォルトで、「主バッファー」にデータを記録します。主バッファーとバッファーポリシーの詳細については、第 11 章「バッファーとバッファリング」を参照してください。

デフォルトアクション

節には、任意の数のアクションと変数操作を含めることができます。節に何も指定しないと、「デフォルトアクション」が実行されます。デフォルトアクションでは、有効なプローブ ID (EPID) が主バッファー内にトレースされます。EPID は、特定の述語とアクションによる特定のプローブの特定の有効化を識別します。DTrace コンシューマは、この EPID から、アクションを引き起こしたプローブを特定できます。実際、データのトレース時には必ず、EPID を指定して、コンシューマにトレース対象のデータを伝える必要があります。デフォルトアクションが EPID のトレースのみになっているのは、このためです。

デフォルトアクションを使用すると、dtrace(1M) コマンドの使用が簡単になります。たとえば、次のコマンド例では、タイムシェアリング (TS) スケジューリングモジュール内のすべてのプローブが有効化され、デフォルトアクションが実行されます。

```
# dtrace -m TS
```

このコマンドを実行すると、次のような出力が得られます。

```
# 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
...
```

データ記録アクション

DTrace のコアアクションは、データ記録アクションで構成されています。これらのアクションのデフォルトの動作は、主バッファーへのデータの格納ですが、投機バッファーへのデータの格納も可能です。主バッファーの詳細については、第 11 章「バッファーとバッファリング」を参照してください。投機バッファーについては、第 13 章「投機トレース」を参照してください。この節では「指定バッファー」という語を使用して、データの記録先が主バッファーか投機バッファー (アクションが `speculate()` の後ろに指定されている場合) であることを示しています。

trace()

`void trace(`*expression*`)`

もっとも基本的なアクションは、D 式 expression を引数とし、指定バッファーに結果をトレースする `trace()` アクションです。以下に、`trace()` アクションの例を示します。

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

tracemem()

`void tracemem(`*address*`, size_t` *nbytes*`)`

`tracemem()` アクションは、最初の引数として D 式 *address*、2 番目の引数として定数 *nbytes* をとります。`tracemem()` は、*address* で指定されたアドレスから *nbytes* で指定された長さのメモリーを、指定バッファーにコピーします。

printf()

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

`printf()` アクションは、`trace()` と同じように、D 式をトレースします。ただし、`printf()` では、複雑な `printf(3C)` 形式の書式が使用されます。`printf(3C)` の場合と同じく、パラメータは *format* 文字列と任意の数の引数です。デフォルトでは、これらの引数が指定バッファーにトレースされます。その後、指定された書式設定文字列に従って、これらの引数に `dtrace(1M)` の出力書式が設定されます。たとえば、133 ページの「`trace()`」 の最初の 2 つの `trace()` の例を、単一の `printf()` 内で組み合わせて使用すると、以下のようになります。

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

`printf()` の詳細については、第 12 章「出力書式」を参照してください。

printa()

`void printa(`*aggregation*`)`
`void printa(string` *format*`,` *aggregation*`)`

printa() アクションでは、集積体に書式を設定して表示できます。集積体の詳細については、第 9 章「集積体」を参照してください。*format* を省略した場合、printa() は、*aggregation* で指定された集積体の処理を行い、デフォルトの書式で表示せよという DTrace コンシューマへの指令をトレースする以外に、何も行いません。*format* を指定した場合、集積体に指定された書式が設定されます。printa() の書式文字列については、第 12 章「出力書式」を参照してください。

printa() は、DTrace コンシューマに集積体を処理させる指令をトレースするだけです。カーネル内で集積体の処理を行うことはありません。したがって、printa() のトレース指令から実際に指令が処理されるまでの時間は、バッファー処理に影響を及ぼす要因によって変化します。こうした要因には、集積間隔、バッファリングポリシー、バッファーの切り替え間隔 (バッファリングポリシーが switching である場合) などがあります。これらの要因の詳細については、第 9 章「集積体」と第 11 章「バッファーとバッファリング」を参照してください。

stack()

```
void stack(int nframes)
void stack(void)
```

stack() アクションは、カーネルスタックトレースを指定バッファーに記録します。カーネルスタックの深さは、*nframes* で指定します。*nframes* を省略した場合、stackframes オプションで指定された数のスタックフレームが記録されます。次に例を示します。

```
# 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
   ...
```

stack() アクションは、集積体キーとしても使用できるという点で、その他のアクションとは少し異なっています。

```
# 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 nframes, int strsize)
void ustack(int nframes)
void ustack(void)
```

ustack() アクションは、指定バッファーにユーザースタックトレースを記録します。ユーザースタックの深さは、*nframes* で指定します。*nframes* を省略した場合、ustackframes オプションで指定された数のスタックフレームが記録されます。ustack() では、プローブ起動時の呼び出しフレームのアドレスを特定できますが、DTrace コンシューマがユーザーレベルで ustack() アクションを処理しないかぎり、スタックフレームはシンボルに翻訳されません。*strsize* にゼロ以外の値が指定されている場合、ustack() は指定された量の文字列空間を割り当て、これを使ってカーネルから直接、アドレスからシンボルへの翻訳を行います。このユーザーシンボルの直接翻訳機能は、現在、Java 仮想マシンのバージョン 1.5 以降でしか提供されていません。Java のアドレスからシンボルへの翻訳が行われると、Java クラスとメソッド名を持つ Java フレームが含まれているユーザースタックに、注釈が付けられます。翻訳できなかったフレームは、16 進アドレスで表されます。

以下の例では、文字列空間を持たないスタックをトレースしているため、Java のアドレスからシンボルへの翻訳は行われません。

```
# 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
```

Java 仮想マシンの C と C++ のスタックフレームは、C++ 符号化シンボル名でシンボリックに表現されます。Java スタックフレームは、16 進アドレスで表現されます。以下は、ゼロでない文字列空間で `ustack()` を呼び出す例です。

```
# 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
```

この出力例から、Java スタックフレームのシンボリックなスタックフレーム情報を確認できます。いくつかの関数が静的関数で、アプリケーションシンボルテーブルにエントリがないので、この出力にはまだ 16 進フレームが含まれています。これらのフレームを翻訳することはできません。

Java フレーム以外のフレームの `ustack()` シンボル翻訳は、スタックデータの記録後に行われます。このため、シンボル翻訳の実行前に対応するユーザープロセスが終了してしまい、スタックフレーム翻訳ができない可能性があります。シンボル翻訳の前にユーザープロセスが終了した場合、次の例のように、`dtrace` からの警告メッセージと、16 進スタックフレームのリストが表示されます。

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

この問題に対処するためのテクニックについては、第 33 章「ユーザープロセスのトレース」を参照してください。

最後に、事後 DTrace デバッガコマンドはフレーム翻訳を実行できないので、`ring` バッファーポリシーで `ustack()` を使用すると、常に生の `ustack()` データが返されることになります。

以下の D プログラムの例では、`ustack()` が使用されていますが、*strsize* の指定は省略されています。

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

デフォルトの Solaris インストールで、Netscape の Web ブラウザ `.netscape.bin` を指定してこのプログラムを実行するには、次のコマンドを使用します。

```
# 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 nframes, int strsize)
void jstack(int nframes)
void jstack(void)
```

jstack() は ustack() の別名で、jstackframes オプションでスタックフレーム数を指定し、jstackstrsize オプションで文字列空間サイズを指定します。デフォルトでは、jstacksize はゼロ以外の値になります。つまり、jstack() を使用すると、Java フレーム翻訳が適切に行なわれるようにスタックがトレースされます。

破壊アクション

一部の DTrace アクションは、明確な方法でシステムの状態に変更を加えることから、破壊アクションと呼ばれます。破壊アクションは、明示的に有効にしないと使用できません。dtrace(1M) に -w オプションを指定すると、破壊アクションを有効にできます。dtrace(1M) で、破壊アクションを明示的でない方法で有効にしようとすると、dtrace は失敗し、次のようなメッセージが表示されます。

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

プロセス破壊アクション

一部の破壊アクションは、特定のプロセスだけに影響を及ぼします。こうしたアクションを実行できるのは、dtrace_proc 権限か dtrace_user 権限を持つユーザーだけです。DTrace のセキュリティー権限については、第 35 章「セキュリティー」を参照してください。

stop()

```
void stop(void)
```

stop() アクションは、有効なプローブを起動するプロセスが次にカーネルを出るとき、proc(4) アクションを使用したときのように強制的に停止します。stop() アクションによって停止させられたプロセスを再開するには、prun(1) ユーティリティーを使用します。stop() アクションでは、任意の DTrace プローブポイントで、プロセスを停止できます。このアクションを使用すると、単純なブレークポイントでは捕捉するのが難しい特定の状態のプログラムを捕捉し、そのプロセスに mdb(1) などの従来のデバッガを接続できます。gcore(1) ユーティリティーを使用すれば、停止したプロセスの状態をあとで分析できるようにコアファイルに保存することもできます。

raise()

```
void raise(int signal)
```

raise() アクションは、現在実行中のプロセスに、指定されたシグナルを送信します。このアクションは、kill(1) コマンドでプロセスにシグナルを送る処理とよく似ています。raise() アクションでは、プロセスの実行中の特定のポイントでシグナルを送信できます。

copyout()

```
void copyout(void *buf, uintptr_t addr, size_t nbytes)
```

copyout() アクションは、指定されたバッファー (*buf*) から、現在のスレッドに関連付けられたプロセスのアドレス空間内の指定されたアドレス (*addr*) へ、指定されたバイト数 (*nbytes*) をコピーします。ユーザー空間アドレスが、現在のアドレス空間内の有効なフォルトインページに対応していない場合、エラーが生成されます。

copyoutstr()

```
void copyoutstr(string str, uintptr_t addr, size_t maxlen)
```

copyoutstr() アクションは、指定された文字列 (*str*) を、現在のスレッドに関連付けられたプロセスのアドレス空間内の指定されたアドレス (*addr*) へコピーします。ユーザー空間アドレスが、現在のアドレス空間内の有効なフォルトインページに対応していない場合、エラーが生成されます。文字列長は、strsize オプションで設定された値までになります。詳細については、第 16 章「オプションとチューニング可能パラメータ」を参照してください。

system()

```
void system(string program, ...)
```

system() アクションは、*program* で指定されたプログラムを、シェルで実行したときのようにして実行します。*program* 文字列には、printf() または printa() の書式変換を含めることができます。その書式変換に合った引数を指定する必要があります。有効な書式変換の詳細については、第 12 章「出力書式」を参照してください。

以下は、毎秒 1 回 date(1) コマンドを実行する例です。

```
# 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
```

以下は、先ほどの例よりもアクションの使い方が複雑になった例です。この例では、*program* 文字列内で、printf() 変換と、パイプのような従来のフィルタリングツールが使用されています。

```
#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);
}
```

このスクリプトを実行すると、次のような出力が得られます。

```
# ./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
```

指定されたコマンドは、起動するプローブのコンテキストでは実行されません。実行されるのは、`system()` アクションの詳細情報が格納されているバッファーがユーザーレベルで処理されたときです。この処理がどのようにしていつ行われるのかは、バッファリングポリシーによって異なります。詳細については、第 11 章「バッファーとバッファリング」を参照してください。デフォルトのバッファリングポリシーでは、バッファー処理間隔は `switchrate` オプションで指定されます。次の例のように、明示的に `switchrate` をチューニングして、デフォルト値の 1 秒よりも長くした場合、`system()` 内で遅延が発生します。

```
#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);
}
```

このスクリプトを実行すると、次のような出力が得られます。

```
# 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
```

walltime の値は異なっているが、date の値は一致している点に注目してください。これは、date(1) コマンドが、system() アクションが記録されたときではなく、バッファーが処理されたときにだけ実行されるからです。

カーネル破壊アクション

システム全体に影響を及ぼす破壊アクションもあります。これらのアクションは、システム上のすべてのプロセスに影響を及ぼすだけでなく、影響を受けるシステムのネットワークサービスによっては、その他のシステムにも暗黙的または明示的に影響を及ぼします。したがって、これらのアクションを使用するときは、細心の注意を払ってください。

breakpoint()

```
void breakpoint(void)
```

breakpoint() アクションは、カーネルブレークポイントを設定して、システムを停止し、カーネルデバッガに制御を移します。カーネルデバッガは、アクションを引き起こした DTrace プローブを表す文字列を発行します。たとえば、次の内容を実行するとします。

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

SPARC 版 Solaris 環境では、コンソールに次のメッセージが表示されます。

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

x86 版 Solaris 環境では、コンソールに次のメッセージが表示されます。

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

プローブ記述に続くアドレスは、DTrace 内の有効化制御ブロック (ECB) のアドレスです。このアドレスを使って、ブレークポイントアクションを引き起こすプローブ有効化の詳細情報を得ることができます。

`breakpoint()` アクションの操作を間違うと、このアクションが必要以上に何度も呼び出される可能性があります。この問題が発生すると、ブレークポイントアクションを引き起こす DTrace コンシューマを終了することもできなくなってしまう場合があります。そのような場合は、カーネル整数変数 `dtrace_destructive_disallow` を 1 に設定してください。この設定では、マシン上のすべての破壊アクションが拒否されます。この設定は、この特別な状況以外では適用しないでください。

`dtrace_destructive_disallow` の設定方法は、厳密には、使用しているカーネルデバッガによって異なります。SPARC システム上で OpenBoot PROM を使用している場合は、`w!` を使用します。

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

`w?` を使って、変数が設定されたことを確認します。

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

`go` と入力して、処理を続行します。

```
ok go
```

x86 または SPARC システム上で `kmdb(1)` を使用している場合は、書式設定 dcmd であるスラッシュ (`/`) を入力し、続けて 4 バイトの書き込み修飾子 (`W`) を指定します。

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

`:c` と入力して、処理を続行します。

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

続行後、破壊アクションを再度有効にするには、`mdb(1)` を使って `dtrace_destructive_disallow` を明示的にゼロに戻す必要があります。

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

panic()

```
void panic(void)
```

panic() アクションを引き起こすと、カーネルパニックが起こります。このアクションは、必要に応じて強制的にシステムのクラッシュダンプを実行するときに使用します。このアクションをリングバッファリングや事後分析と組み合わせることで、問題を把握できます。詳細は、第 11 章「バッファーとバッファリング」と第 37 章「事後トレース」を参照してください。パニックアクションを使用すると、パニックの原因となったプローブを示すパニックメッセージが表示されます。次に例を示します。

```
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) も、リブート時にメッセージを発行します。

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

クラッシュダンプのメッセージバッファーには、プローブのほか、panic() アクションの原因となった ECB も含まれます。

chill()

```
void chill(int nanoseconds)
```

chill() アクションには、DTrace を指定された期間 (*nanoseconds* ナノ秒間) 待つ働きがあります。chill() は、主にタイミング関連の問題の調査に使用します。たとえば、このアクションを使って、競合しているウィンドウを開いたり、定期イベント

を同期させたり、定期イベントの同期を解除したりできます。DTrace プローブコンテキストでは、割り込みは無効になります。このため、chill() を使用すると、割り込み遅延、スケジュール遅延、およびディスパッチ遅延が発生します。このように、chill() は、システム全体に思いがけない影響を及ぼすことがあるので、乱用は避けてください。システムアクティビティは定期的な割り込み処理に依存します。このため、DTrace は、CPU 上で 1 秒間隔のうち 500 ミリ秒間を超えて chill() アクションを実行することを拒否します。最大 chill() 間隔を超過すると、DTrace は、次の例のように、不正な操作を表すエラーを返します。

```
# 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
```

chill() への複数の呼び出しによる合計、または単一のプローブを使用する複数の DTrace コンシューマによる合計が、この時間制限を超えた場合も同様です。たとえば、次のコマンドでも、同じエラーが生成されます。

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

特殊なアクション

この節では、データ記録アクションでも破壊アクションでもないアクションについて説明します。

投機アクション

投機トレース関連のアクションには、speculate()、commit()、および discard() があります。これらのアクションについては、第 13 章「投機トレース」を参照してください。

exit()

```
void exit(int status)
```

exit() アクションでは、トレースをただちに終了できます。また、DTrace コンシューマに、トレースを終了し、最終処理を行い、指定された状態 status で exit(3C) を呼び出すように指示できます。exit() は状態をユーザーレベルに戻すという点でデータ記録アクションに分類されます。ただし、その他のデータ記録アクションと

は異なり、exit() を投機的にトレースすることはできません。exit() は、バッファーポリシーとは関係なく、DTrace コンシューマを終了させます。exit() はデータ記録アクションなので、落とされることがあります。

exit() を呼び出すと、ほかの CPU 上ですでに実行中の DTrace アクションだけが最後まで実行されます。どの CPU 上でも、新しいアクションは実行されません。この規則の唯一の例外は、END プローブの処理です。END プローブは、DTrace コンシューマが exit() アクションを処理し、トレースを終了するように指示したあとで呼び出されます。

サブルーチン

サブルーチンは、通常、DTrace の内部状態にのみ影響を及ぼすという点で、アクションとは異なっています。このため、「破壊サブルーチン」というようなものは存在しません。また、サブルーチンがデータをバッファー内でトレースすることもありません。サブルーチンには、セクション 9F と 3C のインタフェースに類似したものが多数含まれています。対応するサブルーチンの詳細については、Intro(9F) と Intro(3) のマニュアルページを参照してください。

alloca()

```
void *alloca(size_t size)
```

alloca() は、スクラッチ空間から *size* バイトを割り当て、割り当てられたメモリーへのポインタを返します。必ず 8 バイトのバイト列を持つポインタが返されます。スクラッチ空間は、節の開始から完了までの間しか有効ではありません。alloca() で割り当てられたメモリーは、節の完了時に割り当て解除されます。使用できるスクラッチ空間が不足している場合、メモリーの割り当ては行われず、エラーが生成されます。

basename()

```
string basename(char *str)
```

basename() は、basename(1) に相当します。このサブルーチンは、指定された文字列のコピーから成る文字列を生成します。ただし、/ で終わる接頭辞は付きません。返される文字列には、スクラッチメモリーからメモリーが割り当てられます。したがって、節が完了すると、この文字列は無効になります。使用できるスクラッチ空間が不足している場合、basename は実行されず、エラーが生成されます。

bcopy()

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

bcopy() は、*src* がポイントするメモリーから *dest* がポイントするメモリーへ、*size* バイトをコピーします。すべてのコピー元のメモリーはスクラッチメモリーの外部、すべてのコピー先のメモリーはスクラッチメモリーの内部に存在していなければなりません。この条件が満たされない場合、コピーは行われず、エラーが生成されます。

cleanpath()

```
string cleanpath(char *str)
```

cleanpath() は、*str* で指定されたパスのコピーから成る文字列を生成します。ただし、特定の重複要素は排除されます。特に、パス内の「/./」要素は削除され、「/../」要素は縮められます。パス内の「/../」は、シンボリックリンクを考慮せずに縮められます。このため、cleanpath() を使用すると、有効なパスが縮められ、短い無効なパスが返されることがあります。

たとえば、*str* が「/foo/../bar」で、/foo が /net/foo/export のシンボリックリンクになっている場合、bar があるのは / ではなく /net/foo であるのに、cleanpath() は文字列「/bar」を返します。この問題が発生するのは、cleanpath() が起動プローブのコンテキストで呼び出されるからです。起動プローブのコンテキストでは、完全なシンボリックリンク解決は行われず、任意の名前は使用できません。返される文字列には、スクラッチメモリーからメモリーが割り当てられます。したがって、節が完了すると、この文字列は無効になります。使用できるスクラッチ空間が不足している場合、cleanpath は実行されず、エラーが生成されます。

copyin()

```
void *copyin(uintptr_t addr, size_t size)
```

copyin() は、指定されたユーザーアドレス addr から DTrace スクラッチバッファーに、指定されたサイズ size バイトをコピーし、このバッファーのアドレスを返します。ユーザーアドレスは、現在のスレッドに関連付けられたプロセスの空間に含まれるアドレスであると見なされます。最終的なバッファーポインタは、必ず 8 バイトのバイト列を持つことになります。指定されたアドレスは、現在のプロセス内のフォルトインページに対応している必要があります。アドレスがフォルトインページに対応していない場合や、使用できるスクラッチ空間が不足している場合は、NULL が返され、エラーが生成されます。copyin のエラーが発生する可能性を減らすテクニックについては、第 33 章「ユーザープロセスのトレース」を参照してください。

copyinstr()

```
string copyinstr(uintptr_t addr)
```

copyinstr() は、指定されたユーザーアドレス addr から DTrace スクラッチバッファーに、NULL で終了する C 文字列をコピーし、このバッファーのアドレスを返します。ユーザーアドレスは、現在のスレッドに関連付けられたプロセスの空間に含まれるアドレスであると見なされます。文字列長は、strsize オプションで設定された値以下に制限されます。詳細については、第 16 章「オプションとチューニング可能パラメータ」を参照してください。copyin の場合と同じく、指定されたアドレスは、現在のプロセス内のフォルトインページに対応している必要があります。アドレスがフォルトインページに対応していない場合や、使用できるスクラッチ空間が不足している場合は、NULL が返され、エラーが生成されます。第 33 章「ユーザープロセスのトレース」 エラーが発生する可能性を減らすテクニックについては、Chapter 33, User Process Tracingを参照してください。

copyinto()

```
void copyinto(uintptr_t addr, size_t size, void *dest)
```

copyinto() は、指定されたユーザーアドレス addr から、*dest* で指定された DTrace スクラッチバッファーに、指定されたサイズ size バイトをコピーします。ユーザーアドレスは、現在のスレッドに関連付けられたプロセスの空間に含まれるアドレスであると見なされます。指定されたアドレスは、現在のプロセス内のフォルトインページに対応している必要があります。アドレスがフォルトインページに対応していない場合や、コピー先のメモリーの一部がスクラッチ空間内にない場合、コピーは行われず、エラーが生成されます。第 33 章「ユーザープロセスのトレース」 エラーが発生する可能性を減らすテクニックについては、Chapter 33, User Process Tracingを参照してください。

dirname()

```
string dirname(char *str)
```

dirname() は、dirname(1) に相当します。このサブルーチンは、*str* で指定されたパス名の最後のレベルを除くすべてのレベルから成る文字列を生成します。返される文字列には、スクラッチメモリーからメモリーが割り当てられます。したがって、節が完了すると、この文字列は無効になります。使用できるスクラッチ空間が不足している場合、dirname は実行されず、エラーが生成されます。

msgdsize()

```
size_t msgdsize(mblk_t *mp)
```

msgdsize() は、*mp* がポイントしているデータメッセージ内のバイト数を返します。詳細については、msgdsize(9F) のマニュアルページを参照してください。msgdsize() がカウントするのは、M_DATA 型のデータブロックだけです。

msgsize()

```
size_t msgsize(mblk_t *mp)
```

msgsize() は、*mp* がポイントしているメッセージ内のバイト数を返します。msgdsize() がデータのバイト数だけを返すのに対し、msgsize() はメッセージ内の全バイト数を返します。

mutex_owned()

```
int mutex_owned(kmutex_t *mutex)
```

mutex_owned() は、mutex_owned(9F) の実装です。mutex_owned() は、呼び出しスレッドが指定されたカーネル相互排他ロックを所有している場合は、ゼロ以外の値を返します。指定された適応型相互排他ロックの所有者が存在しない場合は、ゼロを返します。

mutex_owner()

```
kthread_t *mutex_owner(kmutex_t *mutex)
```

mutex_owner() は、指定された適応型カーネル相互排他ロックの現在の所有者のスレッドポインタを返します。指定された適応型相互排他ロックの所有者が存在しない場合や、指定された相互排他ロックがスピン相互排他ロックである場合、mutex_owner() は NULL を返します。mutex_owned(9F) のマニュアルページを参照してください。

mutex_type_adaptive()

```
int mutex_type_adaptive(kmutex_t *mutex)
```

mutex_type_adaptive() は、指定されたカーネル相互排他ロックが MUTEX_ADAPTIVE 型 (適応型) の場合はゼロ以外、MUTEX_ADAPTIVE 型でない場合はゼロを返します。次のいずれかの条件が満たされている場合、相互排他ロックは適応型になります。

- 相互排他ロックが静的に宣言されている
- NULL の割り込みブロック Cookie によって生成された 相互排他ロックである
- ハイレベル割り込みに対応していない割り込みブロック Cookie によって生成された 相互排他ロックである

相互排他ロックの詳細については、mutex_init(9F) のマニュアルページを参照してください。Solaris カーネル内のほとんどの 相互排他ロックは適応型です。

progenyof()

```
int progenyof(pid_t pid)
```

progenyof() は、呼び出しプロセス (一致したプローブを引き起こしているスレッドに関連付けられたプロセス) が指定されたプロセス ID pid の子孫にあたる場合、ゼロ以外の値を返します。

rand()

```
int rand(void)
```

rand() は、擬似乱数整数を返します。返される値は、弱い擬似乱数であり、暗号化アプリケーションでは使用できません。

rw_iswriter()

```
int rw_iswriter(krwlock_t *rwlock)
```

rw_iswriter() は、指定された読み取り/書き込みロック rwlock が書き込み側に保持されている場合、または書き込み側から要求されている場合、ゼロ以外の値を返します。ロックが読み取り側だけで保持されていて、書き込み側がブロックされていない場合、またはロックがまったく保持されていない場合、rw_iswriter() はゼロを返します。rw_init(9F) のマニュアルページを参照してください。

rw_write_held()

```
int rw_write_held(krwlock_t *rwlock)
```

w_write_held() は、指定された読み取り/書き込みロック rwlock が書き込み側によって保持されている場合、ゼロ以外の値を返します。ロックが読み取り側だけで保持されている場合やまったく保持されていない場合、rw_write_held () はゼロを返します。rw_init(9F) のマニュアルページを参照してください。

speculation()

```
int speculation(void)
```

speculation() は、speculate() で使用する投機トレースバッファーを予約し、このバッファーの識別子を返します。詳細については、第 13 章「投機トレース」を参照してください。

strjoin()

```
string strjoin(char *str1, char *str2)
```

strjoin() は、*str1* と *str2* の連結から成る文字列を生成します。返される文字列には、スクラッチメモリーからメモリーが割り当てられます。したがって、節が完了すると、この文字列は無効になります。使用できるスクラッチ空間が不足している場合、strjoin は実行されず、エラーが生成されます。

strlen()

```
size_t strlen(string str)
```

strlen() は、指定された文字列 str の長さ (バイト単位) を返します。このとき、終端の NULL バイトは除外されます。

◆ ◆ ◆ 第 11 章

11 バッファーとバッファリング

データのバッファリングと管理は、dtrace(1M) のように、DTrace フレームワークがクライアントに提供する主要なサービスです。この章では、データのバッファリングの詳細と、DTrace のバッファー管理ポリシーを変更するための各種オプションについて説明します。

主バッファー

「主バッファー」は、トレースアクションがデータを記録するデフォルトのバッファーであり、DTrace の呼び出し時に必ず使用されます。トレースアクションに該当するアクションには、以下のものがあります。

exit()	printf()	trace()	ustack()
printa()	stack()	tracemem()	

主バッファーは、常に CPU 単位で割り当てられます。このポリシーはチューニングできません。ただし、cpu オプションを使って、トレースやバッファー割り当てが単一の CPU で行われるようにすることは可能です。

主バッファーのポリシー

DTrace では、カーネル内の高度に制約されたコンテキストでトレースを行うことができます。特に、カーネルソフトウェアが確実にメモリーを割り当てることができないような状況で、トレースを行うことができます。このように、コンテキストに柔軟性があるので、使用できる空間がないときでも、データのトレースが試行される可能性があります。DTrace には、こうした状況に対処するためのポリシーが必要ですが、測定する内容に応じてポリシーを変更したい場合もあります。ときには、新しいデータを破棄するポリシーが必要になることもあります。また、一番古い

データが記録されている空間を再利用して、新しいデータをトレースするポリシーが必要になることもあります。ほとんどの場合、必要とされるのは、使用できる空間がなるべく不足しないようにするポリシーです。こうしたさまざまな要求に対応するため、DTrace では、複数の異なったバッファーポリシーを使用できます。このためには、`bufpolicy` オプションを指定します。このオプションは、コンシューマ単位で設定できます。オプションの設定の詳細については、第 16 章「オプションとチューニング可能パラメータ」を参照してください。

switch ポリシー

主バッファーには、デフォルトで、`switch` バッファーポリシーが設定されています。このポリシーでは、CPU 単位のバッファーのペアが割り当てられます。 ペアの一方はアクティブなバッファー、もう一方はアクティブでないバッファーです。DTrace コンシューマがバッファーを読み取ろうとすると、カーネルはまず、アクティブでないバッファーとアクティブなバッファーの切り替えを行います。バッファーの切り替えは、ウィンドウ内のトレースデータが失われる可能性がない状態で行われます。バッファーの切り替えが完了すると、新たにアクティブでなくなったバッファーが DTrace コンシューマにコピーされます。このポリシーにより、コンシューマは常に、自己矛盾のないバッファーを認識するようになります。 1 つのバッファーのトレースとコピーが同時に行われることはありません。この方法で、ウィンドウ内でのトレースの一時停止や、トレース不能状態の発生を回避することもできます。バッファーの切り替えと読み取りのレートは、`switchrate` オプションにより、コンシューマ側で制御します。 `switchrate` では、その他のレートオプションと同じく、任意の時間接尾辞を指定できます。デフォルトでは、秒当たりのレートが設定されます。`switchrate` とその他のオプションの詳細については、第 16 章「オプションとチューニング可能パラメータ」を参照してください。

注 – 主バッファーをユーザーレベルでデフォルトのレート (1 秒に 1 回) より速いレートで処理するときは、`switchrate` の値を調整してください。主バッファー内の対応するレコードが処理されるときに、ユーザーレベルのアクティビティーを引き起こすアクション (`printa()` や `system()` など) が処理されます。`switchrate` の値によって、そのようなアクションが処理されるレートが決まります。

`switch` ポリシーでは、指定の有効なプローブがアクティブな主バッファー内の使用可能な空間に収まりきらない量のデータをトレースしようとした場合、そのデータが落とされ、CPU 単位の欠落カウントが増分されます。1 回以上欠落が発生すると、`dtrace(1M)` により、次のようなメッセージが表示されます。

```
dtrace: 11 drops on CPU 0
```

合計バッファーサイズより大きいレコードは、バッファーポリシーに関係なく落とされます。欠落の発生を予防するには、`bufsize` オプションで主バッファーのサイズを大きくするか、`switchrate` オプションで切り替えレートを上げます。

switch ポリシーでは、アクティブなバッファーから copyin()、copyinstr()、alloca() にスクラッチ空間が割り当てられます。

fill ポリシー

単一のカーネル内バッファーを使用したい場合もあります。これは、switch ポリシーと適切な D 構造体を使って、D の変数を増分し exit() アクションを適切な述語に付加すれば、実現可能です。しかしこの方法では、欠落が発生する可能性が残ります。単一の大きいカーネル内バッファーを使用し、CPU 単位のバッファーが 1 つ以上占有されるまでトレースを続行するには、fill バッファーポリシーを使用します。このポリシーでは、有効なプローブが主バッファーの残りの空間に収まり切らない量のデータをトレースしようとするまで、トレースが続行されます。空間が足りなくなると、バッファーがいっぱいになったと見なされ、コンシューマに、CPU 単位のバッファーが少なくとも 1 つが占有されたという通知が送られます。dtrace(1M) がいっぱいになったバッファーを検出すると、トレースは停止し、すべてのバッファーが処理されたあと、dtrace は終了します。たとえバッファーに入るデータがあっても、いっぱいになったバッファーにはそれ以上トレースされません。

fill ポリシーを使用するには、bufpolicy オプションに fill を設定します。たとえば、次のコマンドは、バッファーポリシーを fill に設定し、CPU 単位の 2K のバッファーに、すべてのシステムコールエントリをトレースします。

```
# dtrace -n syscall:::entry -b 2k -x bufpolicy=fill
```

fill ポリシーと END プローブ

END プローブは、通常、DTrace コンシューマによってトレースが明示的に停止されるまで起動しません。END プローブは必ず、単一の CPU 上でのみ起動します。ただし、プローブが起動する CPU は未定義です。fill バッファーを使用しているときは、CPU 単位の主バッファーの少なくとも 1 つがいっぱいになったと見なされた時点で、明示的にトレースが停止します。fill ポリシーを選択した場合、END プローブは、いっぱいになったバッファーがある CPU 上でも起動します。fill バッファーへの END のトレースを可能にするため、DTrace は、END プローブが消費する見込みの空間の量を計算し、主バッファーのサイズからこの空間の量を引きます。結果が負の数になった場合、DTrace は起動せず、dtrace(1M) は対応するエラーメッセージを出力します。

```
dtrace: END enablings exceed size of principal buffer
```

予約機構により、いっぱいになったバッファーにも、END プローブ用の空間は必ず残されています。

ring ポリシー

DTrace の ring バッファーポリシーは、障害を引き起こすイベントをトレースするときに役立ちます。障害の再現に時間がかかるときは、最新のデータだけを保存したい場合があります。主バッファーがいっぱいになると、トレースは最初のエントリに戻り、一番古いトレースデータが上書きされます。リングバッファーを使用するには、bufpolicy オプションに文字列 ring を設定します。

```
# dtrace -s foo.d -x bufpolicy=ring
```

dtrace(1M) でリングバッファーを作成する場合は、処理が終了するまで何も出力されません。リングバッファーはその時点で消費され、処理されます。dtrace は、リングバッファーを CPU 順に処理します。CPU のバッファー内のトレースレコードは、古いものから順に並んでいます。switch バッファリングポリシーの場合と同じく、CPU の異なるレコード間の順序付けは行われません。こうした順序付けが必要な場合は、トレース要求の一部として、timestamp 変数をトレースする必要があります。

以下は、#pragma option 指令を使ってリングバッファリングを有効にする例です。

```
#pragma D option bufpolicy=ring
#pragma D option bufsize=16k

syscall::entry
/execname == $1/
{
    trace(timestamp);
}

syscall::rexit:entry
{
    exit(0);
}
```

その他のバッファー

DTrace の有効化では、必ず主バッファーを使用します。一部の DTrace コンシューマは、主バッファーのほかに、カーネル内データバッファーも使用します。この内訳は、「集積体バッファー」 (第 9 章「集積体」 を参照) と、1 個以上の「投機バッファー」 (第 13 章「投機トレース」 を参照) です。

バッファーサイズ

各バッファーのサイズは、コンシューマ単位でチューニングできます。以下の表に示すように、バッファーサイズのチューニング用オプションが別途用意されています。

バッファー	サイズオプション
主体	bufsize
投機	specsize
集積体	aggsize

これらのオプションには、サイズを表す値を設定します。その他のサイズオプションと同じく、値にはサイズ接尾辞 (オプション) を付けることができます。詳細については、第 16 章「オプションとチューニング可能パラメータ」を参照してください。たとえば、dtrace のコマンド行からバッファーサイズを 1M バイトに設定するには、-x を使ってオプションを設定します。

```
# dtrace -P syscall -x bufsize=1m
```

-dtrace に b オプションを指定する方法もあります。

```
# dtrace -P syscall -b 1m
```

また、#pragma D option を使って bufsize を設定することもできます。

```
#pragma D option bufsize=1m
```

選択したバッファーサイズは、各 CPU 上のバッファーのサイズを表します。switch バッファーポリシーでは、bufsize は、各 CPU 上の各バッファーのサイズを表します。デフォルトのバッファーサイズは 4M バイトです。

バッファーのサイズ変更ポリシー

システムのカーネルの空きメモリー容量が不足していて、使用可能なメモリー容量が足りないため、または DTrace コンシューマがチューニング可能な制限値 (第 16 章「オプションとチューニング可能パラメータ」を参照) を超えてしまったため、必要なサイズのバッファーを割り当てられないことがあります。バッファー割り当てエラーのポリシーは、bufresize オプション (デフォルト値は auto) で設定できます。auto バッファーサイズ変更ポリシーでは、正常に割り当てが行われるまで、バッファーのサイズが二等分されます。割り当てバッファーのサイズが要求されたサイズより小さい場合、dtrace(1M) はメッセージを生成します。

```
# dtrace -P syscall -b 4g
dtrace: description 'syscall' matched 430 probes
dtrace: buffer size lowered to 128m
...
```

または

```
# dtrace -P syscall'{@a[probefunc] = count()}' -x aggsize=1g
dtrace: description 'syscall' matched 430 probes
dtrace: aggregation size lowered to 128m
...
```

バッファーの割り当てエラーの発生後、`bufresize` に `manual` を設定して、手動での調整を要求することもできます。このポリシーでは、割り当てエラーが発生すると、DTrace の起動に失敗します。

```
# dtrace -P syscall -x bufsize=1g -x bufresize=manual
dtrace: description 'syscall' matched 430 probes
dtrace: could not enable tracing: Not enough space
#
```

主バッファー、投機バッファー、集積体バッファーを含むすべてのバッファーのサイズ変更ポリシーは、`bufresize` オプションで指定します。

◆ ◆ ◆ 第 12 章

12 出力書式

DTrace の組み込み書式設定関数には、printf() と printa() があります。D プログラム内でこれらの関数を使用することにより、出力書式を設定できます。D コンパイラには、printf(3C) ライブラリルーチンには含まれない機能があるため、printf() の知識をお持ちのユーザーも、必ずこの章に目を通してください。この章では、trace() 関数の書式設定処理と、dtrace(1M) で集積体を表示するときのデフォルトの出力書式についても説明します。

printf()

printf() 関数は、trace() 関数のようにデータをトレースする機能と、データとその他のテキストを指定された書式で出力する機能を兼ね備えています。DTrace は、printf() 関数からの指示を受けて、2 番目以降の各引数に関連付けられたデータをトレースし、最初の printf() 引数 (「書式設定文字列」) で指定された規則に従って結果を書式設定します。

書式設定文字列は、% で始まる任意の数の書式変換を含む標準文字列で、対応する引数の書式を指定します。この例の書式設定文字列の最初の変換は 2 番目の printf() 引数、2 番目の変換は 3 番目の引数 (以降同様) に対応しています。変換と変換の間のすべてのテキストは、変更されずそのまま出力されます。% 変換文字に続く文字は、対応する引数の書式を表します。

DTrace の printf() は、D コンパイラが認識する組み込み関数です。この点で、printf(3C) とは異なります。D コンパイラは、C ライブラリの printf() にはない、次のような便利なサービスを、DTrace の printf() に提供します。

- D コンパイラは、書式設定文字列内の変換の引数を比較します。引数の型と書式変換に互換性がない場合、D コンパイラはエラーメッセージを発行し、問題について報告します。

- D コンパイラは、printf() 書式変換でのサイズ接頭辞の使用を要求しません。C の printf() ルーチンは、引数のサイズの指定を要求します。したがって、%ld (long の場合)、%lld (long long の場合) などの接頭辞を追加する必要があります。D の printf() 文では、これらの接頭辞は不要です。D コンパイラは、引数のサイズと型を識別できます。
- DTrace では、デバッグや監視に役立つ追加の書式文字を使用できます。たとえば、書式変換 %a を使って、ポインタのシンボル名とオフセットを出力できます。

この機能を利用するためには、D プログラム内で、DTrace の printf() 関数の書式設定文字列を、文字列定数として指定しなければなりません。string 型の動的変数は、書式設定文字列として使用できません。

変換指定

書式設定文字列内の変換指定は、パーセント記号 (%) で始まります。その後、順番に次の情報を指定します。

- 変換指定の意味を変更する 0 個以上の「フラグ」。複数指定する場合、どのような順序で指定してもかまいません。詳細は、次の節で説明します。
- 最小「フィールド幅」(オプション)。変換後の値のバイト数が、フィールド幅のバイト数より少ない場合、デフォルトで、値の左側に空白文字が入ります。左詰めフラグ (-) が指定されている場合、空白文字は値の右側に入ります。フィールド幅は、アスタリスク (*) でも指定できます。この場合、フィールド幅は、int 型の追加引数の値に基づいて動的に設定されます。
- 「精度」(オプション)。変換が d、i、o、u、x、および X の場合、表示される最小桁数 (フィールドの先頭にはゼロが入る) を表します。変換が e、E、および f の場合は基数文字の後ろに表示される桁数、変換が g または G の場合は最大有効桁数、変換が s の場合は最大バイト数を表します。精度は、ピリオド (.) とアスタリスク (*)、またはピリオドと 10 進数の文字列で指定します。
- 「サイズ接頭辞」(オプション)。対応する引数のサイズを指定します。詳細については、162 ページの「サイズ接頭辞」を参照してください。D ではサイズ接頭辞を指定する必要はありませんが、C の printf() 関数との互換性のため、使用できるようになっています。
- 「変換指定子」。引数に適用する変換の型を指定します。

printf(3C) 関数では %*n*$ (*n* は 10 進整数) の形式も使用できますが、DTrace の printf() では、この形式の変換指定はサポートされていません。

フラグ指定子

printf() の変換フラグを有効にするには、次の文字を 1 つ以上指定します。複数指定する場合、どのような順序で指定してもかまいません。

’	10 進変換 (%i、%d、%u、%f、%g、%G) の結果の整数部分の書式は、通貨文字を除くさまざまなグループ化文字を使って設定できます。POSIX の C ロケールを含むいくつかのロケールでは、このフラグと併用できる通貨文字以外のグループ化文字は提供されていません。
-	変換の結果は、左詰めになります。このフラグを指定しなければ、右詰めになります。
+	符号付き変換の結果に、常に符号 (+ または -) を付けます。このフラグを指定しなければ、負の値の変換のときだけ符号が付きます。
space	符号付き変換の先頭文字が符号でない場合や、符号付き変換の結果、文字がなくなった場合、結果の前に空白文字を挿入します。space フラグと + フラグの両方が指定された場合、space フラグは無視されます。
#	選択された変換に代替書式が定義されている場合、変換後の値をこの代替書式で表します。変換の代替書式の説明は、変換ごとに記載します。
0	変換が d、i、o、u、x、X、e、E、f、g、および G の場合、符号や基数のあと、フィールド幅に合わせて、先頭にゼロを挿入します。空白文字によるパディングは行いません。0 フラグと - フラグの両方が指定された場合、0 フラグは無視されます。変換が d、i、o、u、x、および X で、精度が指定されている場合、0 フラグは無視されます。0 フラグと ’ フラグの両方が指定された場合、ゼロによるパディングの前にグループ化文字が挿入されます。

幅と精度の指定子

最小フィールド幅は、任意のフラグ指定子と 10 進文字列の組み合わせで指定できます。この場合、フィールド幅は、指定された桁数になります。フィールド幅は、アスタリスク (*) でも指定できます。この場合、フィールド幅は、int 型の追加引数によって決定されます。たとえば、int 変数 w の値からフィールド幅を決定し、このフィールド幅で整数 x を出力する場合、次のような D 文を使用します。

```
printf("%*d", w, x);
```

フィールド幅は、疑問符 (?) でも指定できます。この場合は、オペレーティングシステムカーネルのデータモデル内で、アドレスを 16 進値で表すために必要な文字数に基づいて、フィールド幅が決定されます。たとえば、カーネルが使用しているデータモデルが 32 ビットの場合は 8、64 ビットの場合は 16 になります。

変換の精度は、ピリオド (.) に続く 10 進文字列、またはピリオドに続くアスタリスク (*) で指定できます。アスタリスクを使って精度を指定した場合、変換引数の前に

追加された int 型変数によって精度が決定されます。フィールド幅と精度の両方をアスタリスクで指定した場合、printf() の引数は、幅、精度、値の順で指定する必要があります。

サイズ接頭辞

printf(3C) を使用する ANSI-C プログラムでは、変換引数のサイズと型を指定するため、必ずサイズ接頭辞を使用しなければなりません。D プログラムでは、サイズ接頭辞は不要です。これは、D コンパイラが、printf() 呼び出しごとに変換引数とサイズと型を自動的に指定するためです。D プログラムでも、C との互換性を確保するため、サイズ接頭辞を使用することは可能ですが、なるべく使用しないでください。サイズ接頭辞を使用すると、派生型を使用するとき、コードが特定のデータモデルに結合されてしまいます。たとえば、typedef を、データモデルに基づいて別の整数基本型として再定義する場合、最初のデータモデルと再定義後のデータモデルの両方に対応した C 変換を単独で使用するには、2 つの基本型が明らかになっていなければなりません。さらに、キャスト式を追加するか、複数の書式設定文字列を定義する必要があります。D コンパイラでは、サイズ接頭辞の省略が可能で、引数のサイズが自動的に特定されるので、この問題は起こりません。

サイズ接頭辞は、フラグ、幅、精度の指定子と書式変換名の間 (書式変換名の直前) に挿入します。サイズ接頭辞には、以下のものがあります。

- h (オプション)。d、i、o、u、x、X のいずれかの変換を short または unsigned short に適用します。
- l (オプション)。d、i、o、u、x、X のいずれかの変換を long または unsigned long に適用します。
- ll (オプション)。d、i、o、u、x、X のいずれかの変換を long long または unsigned long long に適用します。
- L (オプション)。e、E、f、g、G のいずれかの変換を long double に適用します。
- l (オプション)。変換 c を引数 wint_t に適用し、変換文字 s を wchar_t 引数のポインタに適用します。

変換書式

各変換文字シーケンスでは、0 個以上の引数のフェッチが行われます。書式設定文字列の引数の指定が不適切な場合や、書式設定文字列がなくなって引数が余ってしまった場合、D コンパイラはエラーメッセージを発行します。未定義の変換書式が指定された場合も、D コンパイラはエラーメッセージを発行します。変換文字シーケンスは、以下のとおりです。

a　ポインタ (uintptr_t 引数) は、カーネルシンボル名 (*module*`*symbol-name* に任意の 16 進バイトオフセットを追加した形式) として出力されます。この値が、既知のカーネルシンボルで定義された範囲にない場合は、16 進整数として出力されます。

c　引数 char、short、int は ASCII 文字として出力されます。

C　引数 char、short、int は、プリント可能 ASCII 文字である場合、ASCII 文字として出力されます。出力可能文字でない場合、表 2–5 のように、対応するエスケープシーケンスを使って出力されます。

d　引数 char、short、int、long、long long は、10 進 (base 10) 整数として出力されます。引数が signed である場合、符号付きの値が出力されます。引数が unsigned である場合、符号なしの値が出力されます。この変換の効果は、i と同じです。

e、E　引数 float、double、long double は、[-]*d*. *ddd*±*dd* (基数文字の前は 1 桁、後ろは精度と同じ桁数) の形式に変換されます。引数がゼロ以外の場合、基数文字もゼロ以外になります。精度を指定しない場合は、デフォルトの精度値は 6 です。精度が 0 で # フラグを指定しない場合は、基数文字は表示されません。E 変換書式では、e ではなく E の付いた指数が生成されます。指数は必ず 2 桁以上になります。値は、適切な桁数で丸められます。

f　引数 float、double、long double は、[-]*ddd*.*ddd* (基数文字の後ろは精度と同じ桁数) の形式に変換されます。精度を指定しない場合は、デフォルトの精度値は 6 です。精度が 0 で # フラグを指定しない場合は、基数文字は表示されません。基数文字が表示されるときは、必ずその前に 1 桁以上の数字が表示されます。値は、適切な桁数で丸められます。

g、G　引数 float、double、long double は、f または e の形式、さらに G 変換文字の場合は E の形式で、出力されます。有効桁数を示す精度も出力されます。明示的な精度が 0 の場合、1 と見なされます。どの形式が使用されるかは、変換対象の値によって異なります。たとえば、e または E 形式は、変換の結果として得られた指数が、-4 より小さいか、または精度の値以上の場合にかぎって、使用されます。結果の小数部分がゼロの場合、この部分は省略されます。基数文字は、その後ろに桁がある場合にしか表示されません。# フラグを指定すると、結果の小数部分のゼロが省略されなくなります。

i　引数 char、short、int、long、long long は、10 進 (base 10) 整数として出力されます。引数が signed である場合、符号付きの値が出力されます。引数が unsigned である場合、符号なしの値が出力されます。この変換の効果は、d と同じです。

o　引数 char、short、int、long、long long は、符号なし 8 進 (base 8) 整数として出力されます。この変換では、signed の引数または unsigned の引数を使用できます。# フラグを指定すると、結果の最初の桁を強制的にゼロにする必要がある場合に、結果の精度が上がります。

p　ポインタ (uintptr_t) 引数は、16 進 (base 16) 整数として出力されます。D では、あらゆる型のポインタ引数を使用できます。# フラグを指定すると、結果がゼロ以外になった場合、その前に 0x が付加されます。

s　引数は、char の配列か string でなければなりません。配列または string のバイトが終端 NULL 文字またはデータの終わりまで読み取られ、解釈されたあと、ASCII 文字として出力されます。精度を省略した場合、精度は無限と見なされます。したがって、最初の NULL 文字までのすべての文字が出力されます。精度を指定した場合、文字配列のうち、対応するスクリーンカラム数で表示される部分だけが出力されます。書式設定の対象の引数が char * 型である場合、string にキャストするか、引数の前に D 演算子 stringof を付加します。この演算子を付加すると、DTrace は、文字列のバイト数だけをトレースし、書式設定を行うようになります。

S　引数は、char の配列か string でなければなりません。引数は、%s 変換の場合と同様にして処理されます。ただし、プリント可能な ASCII 文字でない場合は、表 2–5 のように、対応するエスケープシーケンスで置き換えられます。

u　引数 char、short、int、long、long long は、符号なし 10 進 (base 10) 整数として出力されます。この変換では、signed の引数または unsigned の引数を使用できます。結果は常に unsigned の書式になります。

wc　引数 int はワイド文字 (wchar_t) に変換され、このワイド文字が出力されます。

ws　引数は、wchar_t の配列でなければなりません。配列のバイトが終端 NULL 文字またはデータの終わりまで読み取られ、解釈されたあと、ワイド文字として出力されます。精度を省略した場合、精度は無限と見なされます。したがって、最初の NULL 文字までのすべてのワイド文字が出力されます。精度を指定した場合、ワイド文字配列のうち、対応するスクリーンカラム数で表示される部分だけが出力されます。

x、X　引数 char、short、int、long、long long は、符号なし 16 進 (base 16) 整数として出力されます。この変換では、signed の引数または unsigned の引数を使用できます。x 形式の変換の場合、文字数字 abcdef を使用します。X 形式の変換の場合、文字数字 ABCDEF を使用します。# フラグを指定すると、結果がゼロ以外になった場合、その前に 0x (%x の場合)、または 0X (%X の場合) が付加されます。

Y　引数 uint64_t は、協定世界時の 1970 年 1 月 1 日 00:00 からの経過時間 (ナノ秒数) として解釈され、「%Y %a %b %e %T %Z」のような cftime(3C) 形式で出力されます。協定世界時の 1970 年 1 月 1 日 00:00 からの経過時間 (ナノ秒数) は、walltimestamp 変数で表されます。

%　パーセント記号 (%) を文字どおりに出力します。引数の変換は行われません。変換指定は %% だけです。

printa()

printa() 関数は、D プログラム内の集積体の結果の書式設定に使用します。この関数は、次のいずれかの形式で呼び出します。

```
printa(@aggregation-name);
printa(format-string, @aggregation-name);
```

最初の形式を使用した場合、dtrace(1M) コマンドは、集積体 aggregation-name のデータのスナップショットを取り、集積体のデフォルト出力書式 (第 9 章「集積体」を参照) と同じ書式で結果を出力します。

2 番目の形式を使用した場合、dtrace(1M) コマンドは、集積体 aggregation-name のデータのスナップショットを取り、次の規則に従って *format string* に指定された変換を適用し、その結果を出力します。

- 集積体を作成する際に使用した組署名と一致する書式変換を使用する必要があります。組要素は、それぞれ 1 回ずつ使用できます。たとえば、次の D 文でカウントを集積するとします。

  ```
  @a["hello", 123] = count();
  @a["goodbye", 456] = count();
  ```

  さらに、プローブ節に D 文 printa(*format-string* , @a) を追加すると、dtrace は集積体 aggregation-name のデータのスナップショットを取り、次の文を入力したときと同じ結果を出力します。

  ```
  printf(format-string, "hello", 123);
  printf(format-string, "goodbye", 456);
  ```

  集積体内に定義された各組についても同様です。

- printf() の場合と異なり、printa() で使用する書式設定文字列には、組要素をすべて含める必要はありません。たとえば、長さ 3 の組と、書式変換 1 つだけを使用することも可能です。したがって、printa() 出力では、任意の組キーを省略できます。このためには、集積体の宣言に変更を加えて、省略したいキーを組の末尾に移動し、そのキーに対応する変換指定子を printa() 書式設定文字列から除外します。
- 出力に集積体の結果を含めるには、printa と併用する場合にかぎり有効な、書式設定フラグ文字 @() を追加します。@ フラグは、任意の適切な書式変換指定子と組み合わせて使用できます。また、単一の書式設定文字列内で複数回使用できます。このため、組の結果がどこに出力されるかは決まっておらず、場合によっては複数回出力されることもあります。集積関数と組み合わせて使用することができる変換指定子は、集積関数の結果の型によって決まります。集積体の結果の型は、次のとおりです。

avg()	uint64_t
count()	uint64_t
lquantize()	int64_t
max()	uint64_t
min()	uint64_t
quantize()	int64_t
sum()	uint64_t

たとえば、avg() の結果に書式を設定するには、%d、%i、%o、%u、%x のいずれかの書式変換を適用します。関数 quantize() と lquantize() は、結果を単一の値ではなく、ASCII テーブルとして書式設定します。

以下に、printa() を使用する D プログラムの例を示します。この例では、profile プロバイダを使って caller の値を収集し、結果を単純なテーブルとして書式設定しています。

```
profile:::profile-997
{
    @a[caller] = count();
}

END
{
    printa("%@8u %a\n", @a);
}
```

dtrace でこのプログラムを実行し、しばらく待ってから Control-C キーを押します。すると、次のような出力が得られます。

```
# 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
       ...
```

trace() のデフォルト書式

printf() の代わりに trace() 関数を使ってデータを捕捉する場合、dtrace コマンドを実行すると、結果にデフォルトの出力書式が適用されます。データのサイズが 1、2、4、または 8 バイトである場合、結果は 10 進整数値として書式設定されます。これ以外のサイズで、バイトシーケンスとして解釈される出力可能文字列である場合、データは ASCII 文字列として出力されます。これ以外のサイズで、出力可能文字列でない場合、データは 16 進整数の書式が設定された連続するバイト値として出力されます。

◆ ◆ ◆ 第 13 章

13 投機トレース

この章では、DTrace の「投機トレース」機能について説明します。この機能を利用すると、一時的にデータをトレースし、このデータをトレースバッファーに「コミット」するか「破棄」するかをあとで決めることができます。DTrace では、重要でないイベントを除去する手段として、主に「述語」(第 4 章「D プログラムの構造」を参照) を使用します。述語は、そのプローブイベントがユーザーにとって重要かどうか、プローブの起動時にわかっている際に有用です。たとえば、特定のプロセスのアクティビティや特定のファイル記述子のアクティビティだけが重要である場合は、プローブの起動時に、それが該当のプロセスやファイル記述子に関係したプローブかどうか判断が可能です。一方、プローブの起動後しばらくしてからでないと、そのプローブイベントが重要かどうかわからない場合もあります。

たとえば、あるシステムコールが一般的なエラーコード (`EIO` や `EINVAL` など) を出力して異常終了する場合に、エラー条件の原因となっているコードパスを調べるとします。該当のコードパスを探すため、すべてのプローブを有効にすることも考えられます。ただしこの方法は、意味のある述語を作成できる程度に問題のシステムコールが特定されている場合にかぎります。問題が散発的であったり特定が難しい場合は、今後重要になる可能性があるイベントをすべてトレースし、あとで、問題のコードパスに無関係なデータをふるい落とさなければなりません。この場合、本当に重要なイベントはわずかであっても、大量のイベントをトレースする必要があるので、後処理が難しくなります。

投機トレース機能は、こうした状況で利用します。この機能では、1 つ以上のプローブ位置で一時的にデータをトレースしたあと、データを主バッファーにコミットするかどうかは、別のプローブ位置で決めることができます。結果的に、重要な情報だけがトレースデータとして確保されます。さらに、後処理が不要になり、DTrace のオーバーヘッドも最小限に抑えることができます。

投機インタフェース

以下の表に、DTrace 投機関数を一覧します。

表 13–1　DTrace 投機関数

関数名	引数	説明
`speculation`	なし	新しい投機バッファーの識別子を返す
`speculate`	ID	同一節内の残りの部分を、指定された ID の投機バッファーにトレースする
`commit`	ID	指定された ID の投機バッファーをコミットする
`discard`	ID	指定された ID の投機バッファーを破棄する

投機の作成

`speculation()` 関数は、投機バッファーを割り当て、投機識別子を返します。返された投機識別子は、その後、`speculate()` 関数を呼び出すときに使用します。投機バッファーは有限のリソースです。 `speculation()` の呼び出し時に使用可能な投機バッファーがない場合、ID としてゼロが返され、対応する DTrace エラーカウンタの値が大きくなります。値がゼロの ID は常に無効ですが、`speculate()`、`commit()`、`discard()` のいずれかの関数に渡すことができます。`speculation()` の呼び出しに失敗した場合、次のような `dtrace` メッセージが表示されます。

```
dtrace: 2 failed speculations (no speculative buffer space available)
```

投機バッファーの数は、デフォルトでは 1 個ですが、さらに増やすこともできます。詳細については、177 ページの「投機オプションとチューニング」を参照してください。

投機の使用

投機を使用するときは、`speculation()` から返される識別子を、節内のデータ記録アクションの前に `speculate()` 関数に渡す必要があります。1 つの節内で、`speculate()` 以降に含まれるデータ記録アクションはすべて、投機的にトレースされます。1 つの D プローブ節内で、データ記録アクションのあとに `speculate()` を呼び出すと、D コンパイラから、コンパイル時エラーが返されます。1 つの節には必ず、投機トレース要求だけ、またはそれ以外のトレース要求だけを含めるようにしてください。同じ節に、投機トレース要求とそれ以外のトレース要求の両方を含めることはできません。

集積アクション、破壊アクション、`exit` アクションは、投機的に処理することはできません。これらのいずれかのアクションを `speculate()` と同じ節に含めると、コン

パイル時エラーが発生します。同じ節で、複数の speculate() を使用することはできません。speculate() は、節当たり 1 個と決まっています。speculate() 1 個以外に何も含まれない節では、デフォルトのアクション (プローブ ID が有効なプローブのみをトレースする) が投機的にトレースされます。デフォルトアクションについては、第 10 章「アクションとサブルーチン」を参照してください。

通常、speculation() の結果は、スレッド固有変数に割り当てます。その後は、このスレッド固有変数を、ほかのプローブの述語や speculate() の引数として使用します。次に例を示します。

```
syscall::open:entry
{
    self->spec = speculation();
}

syscall:::
/self->spec/
{
    speculate(self->spec);
    printf("this is speculative");
}
```

投機のコミット

投機のコミットは、commit() 関数を使って行います。投機バッファーをコミットすると、そのデータが主バッファーにコピーされます。指定した投機バッファーに、主バッファーには収まり切らない量のデータが含まれている場合、データはコピーされず、バッファーの欠落カウントの値が大きくなります。バッファーを複数の CPU 上で投機的にトレースした場合、コミットされた CPU 上の投機データはすぐにコピーされますが、その他の CPU 上の投機データは、commit() の実行後しばらくしてからコピーされます。したがって、ある CPU 上で commit() の実行が開始されてから、投機バッファー内のデータがすべての CPU 上の主バッファーにコピーされるまでには、少し時間がかかります。ただし、クリーンアップレートで指定された時間より長くなることはありません。詳細については、177 ページの「投機オプションとチューニング」を参照してください。

コミットされた投機バッファーをその後の speculation() 呼び出しで使用することはできません。使用したい場合は、各 CPU の投機バッファーを対応する CPU の主バッファーに完全にコピーする必要があります。同様に、コミットされたバッファーに対するその後の speculate() 呼び出しは、サイレントモードで破棄されます。したがって、その後の commit() 呼び出しや discard() 呼び出しは、何のメッセージも出さずに異常終了します。結局、commit() が含まれる節に、データ記録アクションを含めることはできません。しかし、同じ節に複数の commit() 呼び出しを含めて、複数のバッファーをばらばらにコミットすることは可能です。

投機の破棄

投機を破棄するには、`discard()` 関数を使用します。投機バッファーを破棄すると、その内容が失われます。投機が `discard()` を呼び出した CPU 上でのみアクティブになっている場合は、破棄された投機バッファーをその後の `speculation()` 呼び出しですぐに使用できます。投機が複数の CPU 上でアクティブになっている場合は、`discard()` を呼び出したあと、破棄された投機バッファーをその後の `speculation()` 呼び出しで使用できるようになるまで、少し時間がかかります。ある CPU 上で `discard()` が呼び出されたあと、破棄された投機バッファーをその後の speculation で使用できるようになるまでの時間が、クリーンアップレートで指定された時間より長くなることはありません。`speculation()` を呼び出したときに、すべての投機バッファーが破棄またはコミットされた状態で、使用可能なバッファーがまったく存在しない場合は、次のような `dtrace` メッセージが表示されます。

```
dtrace: 905 failed speculations (available buffer(s) still busy)
```

投機バッファー数やクリーンアップレートをチューニングすれば、すべてのバッファーが同時に使用不能状態になるのを防ぐことができます。詳細については、177 ページの「投機オプションとチューニング」を参照してください。

投機の例

投機は、特定のコードパスを明らかにするために使用できます。以下の例では、`open()` が異常終了したときだけ、open(2) システムコールのコードパスをすべて表示できます。

例 13–1　`specopen.d`: 異常終了した open(2) のコードフロー

```
#!/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.
     */
    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
```

例 13–1 specopen.d: 異常終了した open(2) のコードフロー (続き)

```
     * 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;
}

syscall::open:return,
syscall::open64:return
/self->spec && errno == 0/
{
    /*
     * If errno is not set, we discard the speculation.
```

例 13–1 specopen.d: 異常終了した open(2) のコードフロー (続き)

```
     */
    discard(self->spec);
    self->spec = 0;
}
```

このスクリプトを実行すると、次のような出力が得られます。

```
# ./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
```

投機オプションとチューニング

投機トレースアクションの実行時に投機バッファーがいっぱいになっていると、このバッファーには何のデータも格納されず、欠落カウントの値が大きくなります。この場合、次のような dtrace メッセージが表示されます。

```
dtrace: 38 speculative drops
```

投機欠落が起こっても、投機バッファーのコミット時には、バッファーの内容がすべて主バッファーにコピーされます。同様に、破棄された投機バッファーで欠落が発生した場合でも、投機欠落が発生します。投機欠落を防ぐには、specsize オプションを使って、投機バッファーのサイズを大きくします。specsize オプションには、任意のサイズ接尾辞を付けることができます。投機バッファーのサイズ変更ポリシーは、bufresize オプションで指定します。

speculation() の呼び出し時に、投機バッファーが使用不能になっていることもあります。まだコミットされておらず、破棄されてもいないバッファーが存在する場合、次のような dtrace メッセージが表示されます。

```
dtrace: 1 failed speculation (no speculative buffer available)
```

この種の投機の失敗を防ぐには、nspec オプションを使って、投機バッファー数を増やします。nspec のデフォルト値は 1 です。

すべての投機バッファーがビジー状態になっている場合も、speculation() は失敗します。この場合、次のような dtrace メッセージが表示されます。

```
dtrace: 1 failed speculation (available buffer(s) still busy)
```

ある投機バッファーの commit() が呼び出されたあと、このバッファーの内容がまだすべての CPU でコミットされていないのに speculation() が呼び出されたことがわかります。この種の投機の失敗を防ぐには、cleanrate オプションを使って、CPU のクリーンアップレートの値を大きくします。cleanrate のデフォルト値は 101hz です。

注 – cleanrate オプションの値は、1 秒あたりの回数で指定する必要があります。接尾辞 hz を使用します。

◆ ◆ ◆ 第 14 章

dtrace(1M) ユーティリティー

dtrace(1M) は、DTrace 機能の汎用フロントエンドとして機能します。このコマンドは、D 言語のコンパイラを呼び出すシンプルなインタフェースを備えています。このコマンドでは、DTrace のカーネル機構からバッファーに格納されたトレースデータを取り出したり、一連の基本的なルーチンを使って、トレースデータに書式を設定して出力できます。この章では、dtrace コマンドの全般的なリファレンス情報を提供します。

説明

dtrace コマンドでは、単一の汎用インタフェースを利用して、DTrace が提供する重要なサービス全部にアクセスできます。このコマンドには、次のようなオプションがあります。

- DTrace によって現在公開されているプローブとプロバイダのリストを表示するオプション
- 任意のプローブ記述指定子 (provider、module、function、name) を使って、プローブを直接有効にするオプション
- D コンパイラを実行して、1 つ以上の D プログラムファイル、またはコマンド行に直接入力されたプログラムをコンパイルするオプション
- 匿名トレースプログラムを生成するオプション (第 36 章「匿名トレース」を参照)
- プログラムの安定性に関するレポートを生成するオプション (第 39 章「安定性」を参照)
- DTrace のトレースとバッファリングの動作に変更を加え、新しい D コンパイラ機能を利用できるようにするオプション (第 16 章「オプションとチューニング可能パラメータ」を参照)

インタプリタファイルを作成するために #! で始まる宣言内で dtrace を使用すると、D スクリプトを作成できます (第 15 章「スクリプトの作成」を参照)。dtrace を

使って、D プログラムのコンパイルを試行し、そのプロパティを特定することもできます。この処理は、以下で説明する -e オプションを指定してトレースを有効にすることなしにできます。

オプション

dtrace コマンドには、以下のオプションを指定できます。

dtrace [-32 | -64] [-aACeFGHlqSvVwZ] [-b *bufsz*] [-c *cmd*] [-D *name* [=*def*]] [-I *path*] [-L *path*] [-o *output*] [-p *pid*] [-s *script*] [-U *name*] [-x *arg* [=*val*]] [-Xa | c | s | t] [-P *provider* [[述語]アクション]] [-m [[*provider:*]*module* [[述語]アクション]]] [-f [[*provider:*]*module:*]*func* [[述語]アクション]] [-n [[[*provider:*]*module:*]*func:*]*name* [[述語]アクション]] [-i *probe-id* [[述語]アクション]]

すでに説明した D 言語の構文に従って、*predicate* は任意の D 述語をスラッシュ (//) で囲んだ形式、*action* は任意の D 文のリストを中括弧 ({ }) で囲んだ形式で指定します。オプション -P、-m、-f、-n、または -i の引数として D プログラムコードを指定する場合、シェルによる解釈処理を防ぐため、このプログラムコードのテキストを適切な引用記号で囲む必要があります。指定できるオプションは、次のとおりです。

-32、-64　D コンパイラは、オペレーティングシステムカーネルのネイティブデータモデルを使ってプログラムを生成します。現在のオペレーティングシステムのデータモデルを特定するには、isainfo(1) -b コマンドを実行します。-32 を指定すると、D コンパイラは、32 ビットデータモデルに従って、D プログラムをコンパイルするようになります。-64 を指定すると、D コンパイラは、64 ビットデータモデルに従って、D プログラムをコンパイルするようになります。通常、これらのオプションを指定する必要はありません。オプションを省略しても、dtrace は、自動的にネイティブデータモデルを選択します。データモデルは、整数型のサイズとその他の言語プロパティに影響を及ぼします。D プログラムがどのデータモデル用にコンパイルされたものであっても、32 ビットと 64 ビットの両方のカーネルで実行できます。オプション -32 と -64 では、-G オプションを指定したとき生成される ELF ファイルの形式 (ELF32 または ELF64) も指定できます。

-a　匿名トレースの状態を要求し、トレースデータを表示します。-a オプションと -e オプションを併用した場合、dtrace は、匿名トレース状態の終了後、新しいデータを待たずにただちに終了します。匿名トレースの詳細については、第 36 章「匿名トレース」を参照してください。

-A　匿名トレース用として、driver.conf(4) 指令を生成します。-A オプションを指定した場合、dtrace は、すべての D プログラム (-s オプションで指定されたものとコマンド行で指定されたものを含む) をコンパイル

し、dtrace(7D) 構成ファイル指令のセットを作成して、指定された匿名トレース (第 36 章「匿名トレース」を参照) 用のプローブを有効にしてから、終了します。デフォルトでは、dtrace は、指令を /kernel/drv/dtrace.conf ファイルに格納します。これ以外の出力ファイルは、-o オプションで指定できます。

-b 主トレースバッファーのサイズ bufsz を設定します。トレースバッファーサイズを指定するときは、サイズ接尾辞として、k、m、g、t のいずれかを追加できます。詳細については、第 36 章「匿名トレース」を参照してください。バッファー容量を割り当てることができない場合、dtrace は、bufresize プロパティの設定内容に従って、バッファーサイズを小さくするか終了するかします。

-c 指定されたコマンド *cmd* を実行し、コマンドの実行が完了したら終了します。コマンド行に -c オプションを 2 つ以上指定した場合、dtrace は、子プロセスが終了するたびに終了状態を報告し、すべてのコマンドが終了した時点で終了します。最初のコマンドのプロセス ID は、マクロ変数 $target により、コマンド行または -s オプションを使って指定された任意のプログラムに渡されます。マクロ変数の詳細については、第 15 章「スクリプトの作成」を参照してください。

-C D プログラムをコンパイルする前に、C プリプロセッサ cpp(1) を適用します。C プリプロセッサのオプションは、-D、-U、-I、および -H で指定します。どの程度まで C 標準に準拠させるかは、-X オプションで指定します。C プリプロセッサの呼び出し時に D コンパイラによって定義されるトークンセットについては、-X オプションの説明を参照してください。

-D -C オプションで有効化された cpp(1) を呼び出すとき、指定された名前 *name* を定義します。等号 (=) と追加の値 *value* が指定されている場合、対応する値が名前として割り当てられます。このオプションは、cpp の呼び出しのたびに -D オプションを渡します。

-e 要求に応じてコンパイルを行い、匿名トレース状態 (-a オプションで指定) が終了したあと、プローブの有効化を行う前に終了します。匿名トレースデータの出力後に終了したい場合は、このオプションと -a オプションを併用します。プログラムを実際に実行せず、対応する計測機能を有効にしないでプログラムがコンパイルされたことを検証したい場合は、このオプションと D コンパイラオプションを併用します。

-f トレースする関数名を指定します。関数名を一覧表示するときは -l オプションを使用します。引数は、*provider:module:function*、*module:function*、*function* のいずれかのプローブ記述形式で指定します。指定しなかったプローブ記述フィールドは空になります。この場合、これらのフィールドの値とは関係なくすべてのプローブが選択されます。*function* 以外の修飾子を指定しなかった場合、対応する *function* を持つすべてのプローブが選択

されます。-f の引数には、後ろに任意の D プローブ節を 1 つ付加できます。コマンド行には、複数の -f オプションを同時に指定できます。

-F 関数の開始 (entry) と終了 (return) を識別することにより、トレース出力をひとつにまとめます。関数の entry プローブの報告はインデントされ、 -> の後ろに出力されます。関数の return プローブの報告はインデント解除され、<- の後ろに出力されます。

-G 組み込み DTrace プログラムが格納された ELF ファイルを生成します。プログラム内に指定された DTrace プローブは、ELF オブジェクト内に格納されます。この ELF オブジェクトは再配置可能で、別のプログラムにリンクできます。-o オプションが指定されている場合、ELF ファイルは、このオペランドの引数として指定されたパス名で保存されます。-o オプションが指定されておらず、DTrace プログラムが *filename*.s という名前のファイルに含まれている場合、ELF ファイルは *file*.o という名前で保存されます。これ以外の場合は、d.out という名前で保存されます。

-H -C オプションで有効化された cpp(1) を呼び出すとき、インクルードファイルのパス名を出力します。このオプションは、cpp 呼び出しのたびに -H オプションを渡します。このため、1 行に 1 つずつパス名が stderr に表示されます。

-i トレースするプローブ ID を指定します。プローブ ID を一覧表示するときは -l オプションを使用します。プローブ ID は、dtrace -l の場合と同じく、10 進整数で指定します。-i の引数には、後ろに任意の D プローブ節を 1 つ付加できます。コマンド行には、複数の -i オプションを同時に指定できます。

-I -C オプションで有効化された cpp(1) を呼び出すときに、#include ファイルの検索パスに指定のディレクトリパス *path* を追加します。このオプションは、cpp の呼び出しのたびに -I オプションを渡します。このオプションで指定されたディレクトリは、デフォルトのディレクトリリストの前の検索パスに追加されます。

-l プローブを有効化しないで、一覧表示します。-l オプションを指定した場合、dtrace は、オプション -P、-m、-f、-n、-i、および -s で指定された内容に一致するプローブのレポートを生成します。これらのオプションが省略されている場合は、すべてのプローブを一覧表示します。

-L DTrace ライブラリの検索パスに、指定のディレクトリパス *path* を追加します。DTrace ライブラリには、D プログラムを作成する際に使用できる一般的な定義が収められています。このオプションで指定された *path* は、デフォルトのライブラリ検索パスの後ろに追加されます。

-m トレースするモジュール名を指定します。モジュール名を一覧表示するときは -l オプションを使用します。引数は、*provider:module* または *module* のプローブ記述形式で指定します。指定しなかったプローブ記述フィール

ドは空になります。この場合、これらのフィールドの値とは関係なくすべてのプローブが選択されます。*module* 以外の修飾子を指定しなかった場合、対応する *module* を持つすべてのプローブが選択されます。`-m` の引数には、後ろに任意の D プローブ節を 1 つ付加できます。コマンド行には、複数の `-m` オプションを同時に指定できます。

`-n` トレースするプローブ名を指定します。プローブ名を一覧表示するときは `-l` オプションを使用します。引数は、*provider:module:function:name*、*module:function:name*、*function:name*、*name* のいずれかのプローブ記述形式で指定します。指定しなかったプローブ記述フィールドは空になります。この場合、これらのフィールドの値とは関係なくすべてのプローブが選択されます。*name* 以外の修飾子を指定しなかった場合、対応する *name* を持つすべてのプローブが選択されます。`-n` の引数には、後ろに任意の D プローブ節を 1 つ付加できます。コマンド行には、複数の `-n` オプションを同時に指定できます。

`-o` *A*、`-G`、`-l` オプションの出力ファイル、またはトレースデータの出力ファイルを、`-output` で指定します。`-A` オプションが指定されていて `-o` オプションが指定されていない場合、デフォルトの出力ファイルは `/kernel/drv/dtrace.conf` になります。`-G` オプションが指定されていて、`-s` オプションに *filename*`.d` の形式で引数が指定されていて、かつ `-o` オプションが指定されていない場合、デフォルトの出力ファイルは *filename*`.o` になります。これ以外の場合、デフォルトの出力ファイルは `d.out` になります。

`-p` 指定のプロセス ID *pid* を取り込み、シンボルテーブルをキャッシュに格納します。その後、このプロセスが完了した時点で終了します。コマンド行に `-p` オプションを 2 つ以上指定した場合、`dtrace` は、プロセスが終了するたびに終了状態を報告し、すべてのコマンドが終了した時点で終了します。最初のプロセス ID は、マクロ変数 `$target` により、コマンド行または `-s` オプションを使って指定された任意の D プログラムに渡されます。マクロ変数の詳細については、第 15 章「スクリプトの作成」を参照してください。

`-P` トレースするプロバイダ名を指定します。プロバイダ名を一覧表示するときは `-l` オプションを使用します。残りのプローブ記述フィールド (module、function、name) は空になります。したがって、これらのフィールドの値とは関係なく、すべてのプローブが選択されます。`-P` の引数には、後ろに任意の D プローブ節を 1 つ付加できます。コマンド行には、複数の `-P` オプションを同時に指定できます。

`-q` 非出力モードに設定します。この場合、`dtrace` はメッセージを抑制し、指定されたオプションや D プログラムと一致するプローブ数などを出力しなくなります。このほか、列ヘッダー、CPU ID、プローブ ID も出力対象か

ら除外され、改行文字が挿入されるだけになります。trace()、printf() などの D プログラム文でトレースされ、書式設定されたデータだけが、stdout に出力されます。

-s 指定された D プログラムソースファイルをコンパイルします。-e オプションが指定されている場合、プログラムのコンパイルは行われますが、計測機能は有効化されません。-l オプションが指定されている場合、プログラムのコンパイルが行われ、一致するプローブのセットが一覧表示されます。しかし、計測機能は有効化されません。-e オプションも -l オプションも指定されていない場合、D プログラムで指定された計測機能が有効化され、トレースが開始されます。

-S D コンパイラの中間コードを表示します。D コンパイラは、D プログラムごとに中間コードのレポートを生成し、stderr に渡します。

-U -C オプションで有効化された cpp(1) を呼び出すとき、指定された名前 *name* を未定義にします。このオプションは、cpp の呼び出しのたびに -U オプションを渡します。

-v 冗長モードに設定します。-v オプションを指定すると、dtrace はプログラム安定性レポートを生成します。このレポートには、指定された D プログラムについて最低限のインタフェースの安定性レベルと依存性レベルが示されます。DTrace の安定性レベルについては、第 39 章「安定性」で詳しく説明します。

-V dtrace でサポートされる最新バージョンの D プログラミングインタフェースを報告します。バージョン情報が stdout に出力されると、dtrace コマンドは終了します。DTrace のバージョン管理機能の詳細については、第 41 章「バージョン管理」を参照してください。

-w -s、-P、-m、-f、-n、-i のいずれかのオプションで指定された D プログラム内で、破壊アクションを使用できるようにします。-w オプションが指定されていない場合、破壊アクションを含む D プログラムをコンパイルすることはできません。また、このプログラムを有効化することもできません。破壊アクションの詳細については、第 10 章「アクションとサブルーチン」を参照してください。

-x DTrace 実行時オプションや D コンパイラオプションを有効化したり、変更したりします。それらのオプションについては、第 16 章「オプションとチューニング可能パラメータ」を参照してください。ブール型のオプションを有効にするときは、その名前を指定します。値を持つオプションを設定するときは、オプション名と値 val を等号 (=) で結びます。

-X -C オプションで有効化された cpp(1) を呼び出すとき、ISO C 標準にどの程度まで準拠させるかを指定します。-X オプションの引数は、次のいずれかの文字になります。どの文字が指定されたかによって、__STDC__ マクロの値や、このマクロが使用されるかどうかが決定されます。

a (デフォルト)	ISO C と K&R の互換性拡張機能。ISO C により、セマンティクスの変更が要求されます。-X オプションが指定されていない場合は、このモードがデフォルトになります。-Xa オプションが指定されている場合、cpp を呼び出したときの定義済みマクロ __STDC__ の値はゼロになります。
c (準拠)	ISO C に完全準拠。K&R C 互換性拡張機能はありません。-Xc オプションが指定されている場合、cpp を呼び出したときの定義済みマクロ __STDC__ の値は 1 になります。
s (K&R C)	K&R C のみ。-Xs オプションが指定されている場合、cpp を呼び出したときのマクロ __STDC__ の値は未定義になります。
t (移行)	ISO C と K&R C の互換性拡張機能。ISO C によるセマンティクスの変更要求はありません。-Xt オプションが指定されている場合、cpp を呼び出したときの定義済みマクロ __STDC__ の値はゼロになります。

-X オプションは、D コンパイラがどのようにして C プリプロセッサを呼び出すかという点にのみ影響を及ぼします。このため、-Xa オプションと -Xt オプションは、D から見ると等価です。どちらのオプションを指定しても、C のビルド環境の設定を簡単に再利用できます。

-X のモードとは関係なく、すべてのモードで次の C プリプロセッサ定義が追加指定され、有効になります。

- __sun
- __unix
- __SVR4
- __sparc (SPARC® システムのみ)
- __sparcv9 (SPARC® システムでの 64 ビットプログラムのコンパイル時のみ)
- __i386 (x86 システムでの 32 ビットプログラムのコンパイル時のみ)
- __amd64 (x86 システムでの 64 ビットプログラムのコンパイル時のみ)
- __*'uname -s'*_*'uname -r'*。__SunOS_5_10 のように、uname の出力の小数点を下線 (_) で置き換える
- __SUNW_D=1

- `__SUNW_D_VERSION=0x`*MMmmmuuu* (*MM* はメジャーリリース番号を表す 16 進数、*mmm* はマイナーリリース番号を表す 16 進数、*uuu* はマイクロリリース番号を表す 16 進数。DTrace のバージョン管理機能の詳細については、第 41 章「バージョン管理」を参照)

`-Z` 一致するプローブがないプローブ記述を許可します。`-Z` オプションを指定しなければ、D プログラムファイル内 (`-s` オプション) やコマンド行 (`-P`、`-m`、`-f`、`-n`、`-i` のいずれかのオプション) で指定されたプローブ記述に、一致する既知のプローブがひとつもない場合、`dtrace` はエラーを出して終了します。

オペランド

`s` オプションで指定された任意の D プログラム、またはコマンド行で指定された任意の D プログラムで使用可能なマクロ変数のセット (`$1`、`$2` など) を定義するときは、`-dtrace` のコマンド行に 0 個以上の引数を追加できます。マクロ変数の詳しい使用法については、第 15 章「スクリプトの作成」で説明します。

終了状態

`dtrace` ユーティリティは、以下の終了値を返します。

0 指定された要求の処理が正常に完了しました。D プログラム要求の場合、終了値 0 は、プログラムのコンパイル、プローブの有効化、または匿名状態の取得が正常に行われたことを表します。`dtrace` は、指定されたトレース要求でエラーや欠落が発生した場合でも 0 を返します。

1 致命的なエラーが発生しました。D プログラム要求の場合、終了値 1 は、プログラムのコンパイルに失敗したか、指定された要求に応じられなかったことを表します。

2 指定されたコマンド行オプションまたは引数が無効です。

◆ ◆ ◆ 第 15 章

15 スクリプトの作成

dtrace(1M) ユーティリティーを使用すると、D プログラムから、シェルスクリプトによく似たインタプリタファイルを作成することができます。このインタプリタファイルは、再利用可能な対話型 DTrace ツールとしてインストールできます。D コンパイラと dtrace コマンドは、D コンパイラの拡張機能である「マクロ変数」のセットを提供しています。DTrace スクリプトは、これらのマクロ変数を使って簡単に作成できます。この章では、マクロ変数の機能を紹介し、持続的なスクリプトを作成するためのヒントを示します。

インタプリタファイル

シェルや、awk(1)、perl(1) などのユーティリティーと同じように、dtrace(1M) でも、実行可能なインタプリタファイルを作成できます。インタプリタファイルの先頭には、以下の形式の行が挿入されます。

#! *pathname arg*

pathname はインタプリタのパス、*arg* は単一の引数 (オプション) です。インタプリタファイルを実行すると、システムにより、指定されたインタプリタが呼び出されます。インタプリタファイル内に指定された *arg* (オプション) は、引数としてインタプリタに渡されます。インタプリタファイルのパスと、インタプリタファイルの実行時に指定されたその他の引数は、インタプリタ引数リストに追加されます。そのため、DTrace インタプリタファイルを作成するときは、最低限これらの引数を指定する必要があります。

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

-s オプションの引数は、インタプリタファイルの実行時に、インタプリタファイルのパス名に変換されます。その後、dtrace によって、このファイルが読み取られ、コンパイル後、実行されます。これは、シェルウィンドウに次のコマンドを入力した場合と同じです。

```
# dtrace -s interpreter-file
```

以下に、dtrace インタプリタファイルの作成例と実行例を示します。次の D ソースコードを記述し、interp.d という名前のファイルに保存してください。

```
#!/usr/sbin/dtrace -s
BEGIN
{
    trace("hello");
    exit(0);
}
```

interp.d ファイルを実行可能にして、実行します。

```
# chmod a+rx interp.d
# ./interp.d
dtrace: script ’./interp.d’ matched 1 probe
CPU     ID                    FUNCTION:NAME
  1      1                           :BEGIN   hello
#
```

ファイルの先頭には、これが指令であることを示す 2 文字 (#!) を必ず挿入してください。文字と文字の間、文字の前後には空白文字を入れないでください。D コンパイラは、インタプリタファイルを処理するとき、この行を自動的に無視します。

dtrace は、getopt(3C) を使ってコマンド行オプションを処理します。このため、複数のオプションを単一のインタプリタ引数として指定できます。たとえば、先ほどの例に -q オプションを追加して、インタプリタ指令を変更してみましょう。

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

複数のオプションを指定する場合、どのブール型オプションよりも後ろに -s オプションを指定します。そうしないと、このあとの引数 (インタプリタファイル名) が -s オプションの引数として処理されません。

インタプリタファイル内に、引数をとるオプションを複数指定する必要がある場合は、すべてのオプションと引数を単一のインタプリタ引数として指定できるとは限りません。この場合は、#pragma D option 指令構文を使ってオプションを設定します。すべての dtrace コマンド行オプションは、#pragma で設定できます (第 16 章「オプションとチューニング可能パラメータ」を参照)。

マクロ変数

D コンパイラには、D プログラムやインタプリタファイルの作成時に使用できる組み込みマクロ変数のセットが定義されています。マクロ変数は、ドル記号 ($) で始まる識別子です。D コンパイラは、入力ファイルの処理時に、この変数を 1 回だけ展開します。以下に、D コンパイラのマクロ変数を示します。

表 15–1　D マクロ変数

名前	説明	参照
`$[0-9]+`	マクロ引数	190 ページの「マクロ引数」を参照してください
`$egid`	実効グループ ID	`getegid(2)`
`$euid`	実効ユーザー ID	`geteuid(2)`
`$gid`	実グループ ID	`getgid(2)`
`$pid`	プロセス ID	`getpid(2)`
`$pgid`	プロセスグループ ID	`getpgid(2)`
`$ppid`	親プロセス ID	`getppid(2)`
`$projid`	プロジェクト ID	`getprojid(2)`
`$sid`	セッション ID	`getsid(2)`
`$target`	ターゲットプロセス ID	192 ページの「ターゲットプロセス ID」を参照してください
`$taskid`	タスク ID	`gettaskid(2)`
`$uid`	実ユーザー ID	`getuid(2)`

マクロ引数 `$[0-9]+` とマクロ変数 `$target` を除くすべてのマクロ変数は、展開されたあと、システム属性 (プロセス ID、ユーザー ID など) を表す整数になります。変数は、展開されたあと、現在の `dtrace` プロセスの属性値か、D コンパイラを実行しているプロセスの属性値になります。

インタプリタファイル内でマクロ変数を使用すると、使用するたびに編集を加える必要がない、持続的な D プログラムを作成できます。たとえば、`dtrace` コマンドによって実行されるものを除くすべてのシステムコールをカウントしたい場合は、以下の例のように、D プログラム節に `$pid` を指定します。

```
syscall:::entry
/pid != $pid/
{
```

```
    @calls = count();
}
```

dtrace コマンドを呼び出すたびにプロセス ID は変化しますが、この節は、常に望ましい結果を返します。

マクロ変数は、D プログラム内の整数、識別子、または文字列の代わりに使用できます。マクロ変数は、入力ファイルの解析時に 1 回だけ (非再帰的に) 展開されます。各マクロ変数は、展開後、独立した入力トークンになります。これらを何らかのテキストと連結して 1 つのトークンにすることはできません。たとえば、$pid の展開後の値が 456 であるとして、次の D コードについて考えてみましょう。

```
123$pid
```

この D コードを展開すると、123 と 456 の 2 つのトークンが隣接する結果になり、構文エラーが返されます。単一の整数トークン 123456 が得られるわけではありません。

展開後のマクロ変数は、プログラム節の冒頭の D プローブ記述内では、隣接テキストと連結されます。たとえば、以下の節では、DTrace プロバイダ pid を使って dtrace コマンドを計測できます。

```
pid$pid:libc.so:printf:entry
{
    ...
}
```

マクロ変数は、各プローブ記述フィールド内で 1 回だけ展開されます。プローブ記述の区切り文字 (:) を含めることはできません。

マクロ引数

D コンパイラは、マクロ変数のセットも提供します。これらのマクロ変数は、dtrace コマンド呼び出しで指定する引数オペランドに対応しています。これらの「マクロ引数」へのアクセスには、組み込み名を使用します。D プログラムファイル名や dtrace コマンドには $0、最初の追加オペランドには $1、2 番目の追加オペランドには $2 (以下同様) を指定します。dtrace の -s オプションを指定した場合、$0 は、このオプションとともに指定する入力ファイルの名前になります。コマンド行に D プログラムを指定した場合は、$0 は、dtrace 自体を実行するとき使用する argv[0] の値になります。

マクロ引数は、そのテキストの形式によって、整数、識別子、または文字列に展開されます。マクロ引数は、すべてのマクロ変数と同じように、D プログラム内の整数、識別子、文字列トークンの代わりに使用できます。以下に示す例はいずれも、適切なマクロ変数を指定すれば、有効な D 式になります。

```
execname == $1    /* with a string macro argument */
x += $1           /* with an integer macro argument */
trace(x->$1)      /* with an identifier macro argument */
```

マクロ引数を使って作成した dtrace インタプリタファイルは、実際の Solaris コマンドと同じように機能し、ユーザーやその他のツールによって指定された情報に従って動作を変えます。たとえば次の D インタプリタファイルは、特定のプロセス ID で実行される write(2) システムコールをトレースします。

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

syscall::write:entry
/pid == $1/
{
}
```

このインタプリタファイルを実行可能ファイルにした場合、コマンド行引数を追加して、$1 の値を指定できます。

```
# chmod a+rx ./tracewrite
# ./tracewrite 12345
```

このコマンド呼び出しでは、プロセス ID 12345 で実行される各 write(2) システムコールがカウントされます。

D プログラムが参照するマクロ引数がコマンド行に指定されていない場合は、プログラムを正常にコンパイルすることができず、エラーメッセージが表示されます。

```
# ./tracewrite
dtrace: failed to compile script ./tracewrite: line 4:
  macro argument $1 is not defined
```

D プログラムに未知のマクロ引数を参照させたい場合は、defaultargs オプションを指定します。defaultargs オプションを指定した場合、未知の引数の値は 0 になります。D コンパイラオプションの詳細については、第 16 章「オプションとチューニング可能パラメータ」を参照してください。コマンド行に、D プログラムが参照しない引数を指定した場合も、D コンパイラはエラーを返します。

マクロ引数の値は、整数、識別子、または文字列の形式になっていなければなりません。それ以外の形式の場合、D コンパイラはエラーを返します。DTrace インタプリタファイルに文字列形式のマクロ引数を指定するときは、シェルが二重引用符や文字列の内容を解釈しないように、引数をさらに単一引用符で囲みます。

```
# ./foo '"a string argument"'
```

D マクロ引数が整数または識別子の形式になっている場合でも、あえて文字列トークンとして解釈したい場合は、マクロ変数またはマクロ引数の名前の前に、ドル記号を 2 つ追加します (例: $$1)。こうすれば、D コンパイラは、引数を「二重引用符で

囲まれた文字列」と見なすようになります。通常の D 文字列エスケープシーケンス (表 2–5 を参照) はすべて、文字列形式のマクロ引数の内部で展開されます。これは、$*arg* や $$*arg* の形式のマクロで参照されている場合でも変わりません。`defaultargs` オプションが指定されている場合、$$*arg* の形式で参照される未知の引数の値は、空文字列 ("") になります。

ターゲットプロセス ID

`dtrace` コマンド行の `p` オプションで選択された、または `-c` オプションで作成された特定のユーザープロセスに適用できるスクリプトを作成したい場合は、`-$target` マクロ変数を使用します。コマンド行に指定された D プログラムや `-s` オプションで指定された D プログラムのコンパイルは、プロセスが作成されるか取り込まれるかして、`$target` 変数がこれらのプロセスのうち最初のプロセスのプロセス ID を表す整数に展開されたあと行われます。たとえば、次の D スクリプトでは、特定の従属プロセスによって実行されるシステムコールの内訳がわかります。

```
syscall:::entry
/pid == $target/
{
    @[probefunc] = count();
}
```

このスクリプトを `syscall.d` という名前のファイルに保存し、次のコマンドを実行すると、`date(1)` コマンドによって実行されるシステムコール数を特定できます。

```
# 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
```

◆ ◆ ◆ 第 16 章

16 オプションとチューニング可能パラメータ

DTrace では、将来的なカスタマイズを見込んで、コンシューマにかなりのレベルの柔軟性が与えられています。DTrace 自体も、特別なチューニングがなるべく必要ないように、妥当なデフォルト値と柔軟なポリシーの下に実装されています。それでも、コンシューマ単位で DTrace の動作をチューニングしなければならない場合があります。この章では、DTrace のオプション、チューニング可能パラメータ、およびこれらを変更するためのインタフェースについて説明します。

コンシューマオプション

DTrace のチューニングが必要なときは、オプションを設定したり有効にしたりします。以下の表に、使用可能なオプションを示します。一部のオプションには、対応する dtrace(1M) コマンド行オプションがあります。

表 16–1 DTrace コンシューマオプション

オプション名	値	dtrace(1M) での別名	説明	参照先
aggrate	*time*		集積体の読み取りレート	第 9 章「集積体」
aggsize	*size*		集積体バッファーサイズ	第 9 章「集積体」
bufresize	auto または manual		バッファーのサイズ変更ポリシー	第 11 章「バッファーとバッファリング」
bufsize	*size*	-b	主バッファーサイズ	第 11 章「バッファーとバッファリング」

表 16–1 DTrace コンシューマオプション (続き)

オプション名	値	dtrace(1M) での別名	説明	参照先
cleanrate	*time*		クリーンアップレート。hz 接尾辞を付けて 1 秒あたりの回数で指定する必要があります。	第 13 章「投機トレース」
cpu	スカラー	-c	トレースを有効にする CPU	第 11 章「バッファーとバッファリング」
defaultargs	—		未知のマクロ引数の参照を許可する	第 15 章「スクリプトの作成」
destructive	—	-w	破壊アクションを許可する	第 10 章「アクションとサブルーチン」
dynvarsize	*size*		動的変数空間のサイズ	第 3 章「変数」
flowindent	—	-F	関数の開始 (entry) をインデントし、その前に -> を付ける。関数の終了 (return) のインデントを解除し、その前に <- を付ける	第 14 章「dtrace(1M) ユーティリティー」
grabanon	—	-a	匿名状態を要求する	第 36 章「匿名トレース」
jstackframes	スカラー		jstack() のデフォルトスタックフレームの数	第 10 章「アクションとサブルーチン」
jstackstrsize	スカラー		jstack() の文字列空間のデフォルトサイズ	第 10 章「アクションとサブルーチン」
nspec	スカラー		投機の数	第 13 章「投機トレース」
quiet	—	-q	明示的にトレースされたデータだけを出力する	第 14 章「dtrace(1M) ユーティリティー」
specsize	*size*		投機バッファーサイズ	第 13 章「投機トレース」

表 16–1 DTrace コンシューマオプション (続き)

オプション名	値	dtrace(1M) での別名	説明	参照先
strsize	*size*		文字列サイズ	第 6 章「文字列」
stackframes	スカラー		スタックフレームの数	第 10 章「アクションとサブルーチン」
stackindent	スカラー		stack() と ustack() の出力をインデントするときに使用する空白文字の数	第 10 章「アクションとサブルーチン」
statusrate	*time*		状態チェックレート	
switchrate	*time*		バッファー切り替えレート	第 11 章「バッファーとバッファリング」
ustackframes	スカラー		ユーザースタックフレームの数	第 10 章「アクションとサブルーチン」

サイズを表す値には、k (キロバイト)、m (メガバイト)、g (ギガバイト)、t (テラバイト) のいずれかの接尾辞を付けることができます。時間を表す値には、ns (ナノ秒)、us (マイクロ秒)、ms (ミリ秒)、s (秒)、hz (秒当たりの回数) のいずれかの接尾辞を付けることができます。

オプションの変更

D スクリプト内にオプションを設定するときは、#pragma D、文字列 option、オプション名を順に指定します。値をとるオプションの場合は、オプション名とオプション値を等号 (=) で結びます。以下に、有効なオプション設定の例を示します。

```
#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
```

dtrace(1M) コマンドでは、-x オプションの引数として、コマンド行でオプションを設定することもできます。次に例を示します。

```
# dtrace -x nspec=4 -x grabanon -x bufsize=2g \
    -x switchrate=10hz -x aggrate=100us -x bufresize=manual
```

無効なオプションを指定すると、dtrace は、オプション名が無効であるというメッセージを表示して終了します。

```
# dtrace -x wombats=25
dtrace: failed to set option -x wombats: Invalid option name
#
```

同様に、オプションに無効な値を指定すると、dtrace は値が無効であるというメッセージを表示します。

```
# dtrace -x bufsize=100wombats
dtrace: failed to set option -x bufsize: Invalid value for specified option
#
```

同じオプションを繰り返し指定した場合、先に指定されたオプションは、あとから指定されたオプションで上書きされます。grabanon など、一部のオプションは設定することだけが可能です。こうしたオプションを設定した場合、あとで設定を解除することはできません。

匿名状態を要求する DTrace コンシューマは、匿名を有効化するためのオプションを使用できます。匿名トレースの有効化については、第 36 章「匿名トレース」を参照してください。

◆ ◆ ◆ 第 17 章

17 dtrace プロバイダ

dtrace プロバイダは、DTrace 自体に関連するプローブを提供します。これらのプローブを使って、トレース開始前に状態を初期化したり、トレースの完了後に状態を処理したり、ほかのプローブで発生した予期せぬ実行エラーを処理したりできます。

BEGIN プローブ

BEGIN プローブは、一番最初に起動するプローブです。すべての BEGIN 節が完了するまで、ほかのプローブは起動しません。BEGIN プローブでは、ほかのプローブ内で使用するすべての状態を初期化できます。以下は、BEGIN プローブを使って、mmap(2) の保護ビットとテキスト表現のマッピングを行う連想配列を初期化する例です。

```
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]);
}
```

BEGIN プローブは、未知のコンテキストで起動します。つまり、stack() や ustack() の出力も、コンテキスト固有の変数の値 (例: execname) も決まっていません。これらの値は、有意な情報であると解釈すべきではありません。BEGIN プローブには、引数は定義されていません。

END プローブ

END プローブは、一番最後に起動するプローブです。このプローブは、ほかのすべてのプローブ節が完了するまで起動しません。このプローブを使って、収集された状態を処理したり、出力に書式を設定したりできます。このため、END プローブでは、printa() アクションがよく使用されます。BEGIN プローブと END プローブを組み合わせると、トレースにかかった合計時間を測定できます。

```
BEGIN
{
    start = timestamp;
}

/*
 * ... other tracing actions...
 */

END
{
    printf("total time: %d secs", (timestamp - start) / 1000000000);
}
```

END プローブのその他の使用方法については、122 ページの「データの正規化」と 165 ページの「printa()」を参照してください。

BEGIN プローブの場合と同じく、END プローブにも引数は定義されていません。END プローブの起動は任意のコンテキストで行われるので、これに依存するプログラムを作成してはいけません。

bufpolicy オプションに値 fill を設定してトレースを行うと、END プローブでトレースされるレコード用に、適切な空間が予約されます。詳細については、155 ページの「fill ポリシーと END プローブ」を参照してください。

注 – exit() アクションは、トレースを停止し、END プローブを起動します。ただし、exit() アクションが呼び出されてから END プローブが起動するまでには、多少の遅延が発生します。この遅延の間、プローブは一切起動しません。プローブが exit() アクションを呼び出したあと、DTrace コンシューマが「exit() が呼び出された」と判断し、トレースを停止するまで、END プローブは起動しません。終了状態のチェックの間隔は、statusrate オプションで設定します。詳細については、第 16 章「オプションとチューニング可能パラメータ」を参照してください。

ERROR プローブ

ERROR プローブは、DTrace プローブの節の実行中、実行時エラーが発生した場合に起動します。たとえば、次の例のように NULL ポインタを間接参照する節があると、ERROR プローブが起動します。

例 17–1 error.d: エラーの記録

```
BEGIN
{
    *(char *)NULL;
}

ERROR
{
    printf("Hit an error!");
}
```

このプログラムを実行すると、次のような出力が得られます。

```
# 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
```

この出力から、ERROR プローブが起動したことがわかります。また、dtrace(1M) は、エラーを報告しています。dtrace は独自の方法で ERROR プローブを有効化してエラーを報告できるようにします。ユーザーは、ERROR プローブを利用して、エラー処理をカスタマイズできます。

ERROR プローブには、次の引数があります。

arg1	エラーの原因となったプローブの有効化されたプローブ ID (EPID)
arg2	障害の原因となったアクションのインデックス
arg3	そのアクションへの DIF オフセット。使用できない場合は -1
arg4	障害の型
arg5	指定された障害の型の値

以下の表に、さまざまな障害の型と、対応する arg5 の値を示します。

arg4 の値	説明	arg5 の意味
DTRACEFLT_UNKNOWN	未知の障害	なし
DTRACEFLT_BADADDR	マップされていないアドレスまたは無効なアドレスのアクセス	アクセスされるアドレス
DTRACEFLT_BADALIGN	境界違反メモリーアクセス	アクセスされるアドレス
DTRACEFLT_ILLOP	不正 (無効) な操作	なし
DTRACEFLT_DIVZERO	整数のゼロ除算	なし
DTRACEFLT_NOSCRATCH	スクラッチ割り当てに必要なスクラッチ空間の不足	なし
DTRACEFLT_KPRIV	カーネルアドレスまたはプロパティのアクセスが試みられたが、そのために必要な権限がない	アクセスされるアドレス。適切でない場合は 0
DTRACEFLT_UPRIV	ユーザーアドレスまたはプロパティのアクセスが試みられたが、そのために必要な権限がない	アクセスされるアドレス。適切でない場合は 0
DTRACEFLT_TUPOFLOW	DTrace 内部パラメータスタックのオーバーフロー	なし

ERROR プローブ内で実行されたアクション自体がエラーの原因になっている場合、エラーは何のメッセージも出さずに落とされます。ERROR プローブが再帰的に呼び出されることはありません。

安定性

以下の表に、dtrace プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	安定	安定	共通
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	安定	安定	共通
引数	安定	安定	共通

◆ ◆ ◆ 第 18 章

18 lockstat プロバイダ

lockstat プロバイダは、ロック競合の統計情報を調べたり、ロックの動作をあらゆる面から理解するために役立つ各種プローブを提供します。lockstat(1M) コマンドは、lockstat プロバイダを使って生のデータを収集する DTrace コンシューマです。

概要

lockstat プロバイダが提供するプローブは、競合イベントプローブと保持イベントプローブの 2 種類です。

「競合イベント」プローブは、同期プリミティブの競合に関するプローブです。このプローブは、スレッドが利用可能なリソースを待機する必要が生じた時点で起動します。一般に Solaris は、競合が起こらないように最適化されているため、競合状態が長時間続くことは考えられません。これらの競合イベントプローブは、実際に競合が発生した場合、その状況を把握するために使用します。競合はめったに発生しないので、競合イベントプローブを有効にしても、通常、パフォーマンスにはさほど影響はありません。

「保持イベント」プローブは、同期プリミティブの獲得や解放などの操作に関するプローブです。保持イベントプローブは、同期プリミティブの操作方法に関するさまざまな問題に答えを出すために使用します。Solaris は、同期プリミティブの獲得/解放を非常に頻繁に行います。その回数は、ビジー状態のシステムで、CPU 当たり毎秒数 100 万回にも及びます。このため、競合イベントプローブより保持イベントプローブを有効にしたときのほうが、プローブの及ぼす影響ははるかに大きくなります。プローブを有効にしたときの影響は甚大になりえますが、決して異常なものではありません。本稼働システムでプローブを有効にしても、まったく問題はありません。

lockstat プロバイダは、Solaris のさまざまな同期プリミティブに相当するプローブを提供します。ここからは、これらのプリミティブとプローブについて説明します。

適応型ロックプローブ

「適応型ロック」は、クリティカルセクションに対する相互排他ロックです。このロックは、ほとんどのカーネルコンテキストで獲得可能です。また、コンテキスト制限がほとんどないので、Solaris カーネル内の同期プリミティブの大部分に対応しています。これらのロックは、競合に対する動作に適応性があります。 スレッドは、獲得しようとする適応型ロックがすでに所有されているとき、ロックを所有しているスレッド (所有スレッド) が CPU 上で実行中であるかどうかを確認します。所有スレッドが別の CPU 上で実行中であれば、このスレッド (ロックを獲得しようとしているスレッド) は「スピン」します。所有スレッドが実行中でなければ、「ブロック」されます。

適応型ロック関連の lockstat プローブは、表 18–1 に示すとおり 4 種類あります。これらの各プローブの `arg0` には、適応型ロックに相当する `kmutex_t` 構造体へのポインタが格納されます。

表 18–1 適応型ロックプローブ

`adaptive-acquire`	適応型ロックの獲得後すぐに起動する保持イベントプローブ。
`adaptive-block`	保持されている適応型相互排他ロックでブロックされたスレッドが再起動し、この相互排他ロックを獲得したあと起動する競合イベントプローブ。`adaptive-acquire` と `adaptive-block` の両方が有効なときは、adaptive-block のほうが先に起動します。1 回のロック獲得で `adaptive-block` と `adaptive-spin` の両方のプローブが起動することもあります。`adaptive-block` の `arg1` には、スリープ時間 (ナノ秒単位) が入ります。
`adaptive-spin`	保持されている適応型相互排他ロックでスピンしたスレッドが、この相互排他ロックを正常に獲得したあと起動する競合イベントプローブ。`adaptive-acquire` と `adaptive-spin` の両方が有効なときは、adaptive-spin のほうが先に起動します。1 回のロック獲得で `adaptive-block` と `adaptive-spin` の両方のプローブが起動することもあります。`adaptive-spin` の `arg1` には、「スピン時間」が入ります。スピン時間とは、ロックが獲得されるまでスピンループで消費されたナノ秒数を示す値です。
`adaptive-release`	適応型ロックの解放後すぐに起動する保持イベントプローブ。

スピンロックプローブ

カーネル内の一部のコンテキストでは、スレッドをブロックできません。該当するコンテキストとして、ハイレベル割り込みコンテキストや、ディスパッチャの状態を操作するコンテキストがあります。これらのコンテキストでは、スレッドをブロックできないため、適応型ロックを使用できません。これらのコンテキストでクリティカルセクションに対する相互排他を実現するには、適応型ロックの代わりに「スピンロック」を使用します。スピンロックは、その名前からわかるように、

競合が発生すると、所有スレッドがロックを解放するまで「スピン」します。スピンロック関連の 3 つのプローブについては、表 18–2 のとおりです。

表 18–2 スピンロックプローブ

spin-acquire	スピンロックの獲得後すぐに起動する保持イベントプローブ。
spin-spin	保持されているスピンロックでスピンしたスレッドが、このスピンロックを正常に獲得したあと起動する競合イベントプローブ。spin-acquire と spin-spin の両方が有効なときは、spin-spin のほうが先に起動します。spin-spin の arg1 には、「スピン時間」が入ります。スピン時間とは、ロックが獲得されるまでスピン状態で消費されたナノ秒数を示す値です。スピンカウントは、スピン時間の比較に利用されます。単独ではほとんど利用されません。
spin-release	スピンロックの解放後すぐに起動する保持イベントプローブ。

スピンロックよりも適応型ロックのほうが一般的です。以下のスクリプトで、2 種類のロックのそれぞれの合計を表示すると、この観察が裏付けられます。

```
lockstat:::adaptive-acquire
/execname == "date"/
{
    @locks["adaptive"] = count();
}

lockstat:::spin-acquire
/execname == "date"/
{
    @locks["spin"] = count();
}
```

このスクリプトの実行を開始したら、別のウィンドウを開いて date(1) コマンドを実行します。DTrace スクリプトを終了すると、次のような出力が得られます。

```
# dtrace -s ./whatlock.d
dtrace: script ’./whatlock.d’ matched 5 probes
^C
spin                                                    26
adaptive                                              2981
```

この出力からわかるように、date コマンドの実行中に獲得されたロックの 99 パーセントは適応型ロックです。*date* コマンドのような単純な処理でも、このように多くのロックが獲得されます。Solaris カーネルに代表されるきわめてスケーラブルなシステムでは、きめ細かいロック処理が要求されます。たくさんのロックを獲得する必要があるのは、このためです。

スレッドロック

「スレッドロック」は、スレッド状態を変更する目的でスレッドをロックするために使用する、特殊なスピンロックです。スレッドロック保持イベントは、スピンロック保持イベントプローブ (spin-acquire と spin-release) を使用します。これに対して、競合イベントは、スレッドロック固有のプローブを独自に備えています。スレッドロック保持イベントプローブを、表 18–3 に示します。

表 18–3 スレッドロックプローブ

thread-spin	スレッドロックに対してスレッドがスピンしたあと起動する競合イベントプローブ。その他の競合イベントプローブと同じように、競合イベントプローブと保持イベントプローブの両方が有効になっている場合、thread-spin は spin-acquire より先に起動します。thread-spin は、実際にロックが獲得される前に起動します。この点は、その他の競合イベントプローブとは異なります。その結果、何回かの thread-spin プローブの起動は、1 回の spin-acquire プローブの起動と同じことになります。

読み取り/書き込みロックプローブ

「読み取り/書き込みロック」は、クリティカルセクション内で、「複数の読み取り」と「単一の書き込み」のいずれか一方を許可します。通常、これらのロックを使用するのは、変更されるより検索される機会のほうが多く、かつクリティカルセクション時間が十分にある構造体です。クリティカルセクション時間が短い場合、読み取り/書き込みロックは、ロックを実装するために使用された共有メモリー上で暗黙的に直列化されます。この場合、読み取り/書き込みロックは、適応型ロックと同じになります。読み取り/書き込みロックの詳細については、rwlock(9F) のマニュアルページを参照してください。

読み取り/書き込みロック関連のプローブは、表 18–4 のとおりです。これらの各プローブの arg0 には、適応型ロックに相当する krwlock_t 構造体へのポインタが格納されます。

表 18–4 読み取り/書き込みロックプローブ

rw-acquire	読み取り/書き込みロックの獲得後すぐに起動する保持イベントプローブ。arg1 には、定数 RW_READER (ロックが読み取りとして獲得された場合) または RW_WRITER (ロックが書き込みとして獲得された場合) が入ります。

表 18–4 読み取り/書き込みロックプローブ (続き)

rw-block	保持されている読み取り/書き込みロックでブロックされたスレッドが再起動し、このロックを獲得したあと起動する競合イベントプローブ。arg1 には、現在のスレッドがロックを獲得するまでのスリープ時間 (ナノ秒) が入ります。arg2 には、定数 RW_READER (ロックが読み取りとして獲得された場合) または RW_WRITER (ロックが書き込みとして獲得された場合) が入ります。arg3 と arg4 には、ブロックの原因に関する情報が入ります。現在のスレッドがブロックされたとき保持されていたロックが「書き込み」だった場合にかぎり、arg3 にはゼロ以外の値が入ります。arg4 には、現在のスレッドがブロックされたときの読み取りカウントが入ります。rw-block と rw-acquire の両方のプローブが有効になっている場合、rw-block のほうが rw-acquire より先に起動します。
rw-upgrade	スレッドが、読み取り/書き込みロックを、読み取り側から書き込み側へ正常に昇格させたあと起動する保持イベント。昇格は、ブロックしないインタフェース rw_tryupgrade(9F) でしか行うことができません。このため、昇格には競合イベントがありません。
rw-downgrade	スレッドが、読み取り/書き込みロックの所有権を、書き込み側から読み取り側へ降格したあと起動する保持イベント。降格は、常に競合なしで正常に行われます。このため、降格には競合イベントがありません。
rw-release	読み取り/書き込みロックの解放後すぐに起動する保持イベントプローブ。arg1 には、定数 RW_READER (解放されたロックが読み取りとして保持されていた場合) または RW_WRITER (解放されたロックが書き込みとして保持されていた場合) が入ります。昇格と降格のため、ロックを獲得したとき、まだこのロックが解放されていない場合もあります。

安定性

以下の表に、lockstat プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	共通
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	共通
引数	発展中	発展中	共通

◆ ◆ ◆ 第 19 章

19 profile プロバイダ

profile プロバイダは、時間ベースの割り込みに関連付けられていて、指定された間隔 (固定) で起動するプローブを提供します。これらのアンカーされていないプローブは、特定の実行ポイントではなく非同期割り込みイベントに関連付けられています。これらのプローブでは、単位時間ごとにシステム状態の標本を収集できます。こうして収集された標本は、システムの動作を推測するのに役立ちます。サンプリングレートが高いときや、サンプリング時間が長いときは、正確な推測が可能です。DTrace アクションを使用すると、profile プロバイダにより、システム内のあらゆるデータの標本を収集できます。たとえば、現在のスレッドの状態、CPU の状態、現在のマシンの命令などの標本を収集できます。

注 – profile プロバイダのプローブからスレッド固有変数にアクセスすることはできません。そのようなプローブで特殊な識別子 self を使ってスレッド固有変数を参照しても、何も出力されません。

---

profile-*n* プローブ

profile-*n* プローブは、各 CPU 上で、固定の間隔で起動します。割り込みレベルが高いという特徴があります。*n* はプローブの起動間隔を表しています。割り込みソースは、毎秒 *n* 回起動します。*n* には、時間の単位を表す接尾辞を付加できます。表 19–1 に、有効な接尾辞とその意味を示します。

表 19–1 有効な時間接尾辞

「サフィックス」	時間の単位
nsec または ns	ナノ秒
usec または us	マイクロ秒

表 19–1 有効な時間接尾辞 *(続き)*

「サフィックス」	時間の単位
msec または ms	ミリ秒
sec または s	秒
min または m	分
hour または h	時間
day または d	日
hz	ヘルツ (秒当たりの回数)

以下の例では、97 ヘルツで起動し、現在実行中のプロセスの情報を収集するプローブを作成します。

```
#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);
}
```

この例をしばらく実行すると、次の例のような出力が得られます。

```
# 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
```

実行中のプロセスの情報は、profile-*n* プロバイダでも収集できます。次の D スクリプト例では、1,001 ヘルツのプロファイルプローブを使って、指定されたプロセスの現在の優先順位に関する情報を収集します。

```
profile-1001
/pid == $1/
{
    @proc[execname] = lquantize(curlwpsinfo->pr_pri, 0, 100, 10);
}
```

このスクリプト例の動作を確認するには、まず、ウィンドウ内に次のコマンドを入力します。

```
$ echo $$
12345
$ while true ; do let i=0 ; done
```

別のウィンドウで D スクリプトを短時間間実行します。このとき、 *12345* を echo コマンドから返された PID に置き換えてください。

```
# 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
```

タイムシェアリングスケジューリングクラスの分布状況が出力されます。シェルプロセスは CPU 上でスピン中なので、システムによる優先順位は常に低い状態です。シェルプロセスの実行頻度がさらに低ければ、優先順位は高くなります。この結果を確認するには、スピン中のシェルで Control-C キーを入力し、先ほどのスクリプトを再度実行します。

```
# dtrace -s ./profpri.d 494621
 dtrace: script ’./profpri.d’ matched 1 probe
```

次に、同じシェルに何か文字を入力します。DTrace スクリプトを終了すると、次のような出力が得られます。

```
ksh
           value  ------------- Distribution ------------- count
              40 |                                         0
              50 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 14
              60 |                                         0
```

シェルプロセスは、CPU 上でスピンせず、ユーザー入力を待ってスリープしていました。このため、実際に実行されたときの優先順位がより高くなっています。

tick-*n* プローブ

tick-*n* プローブは、profile-*n* プローブと同じように、割り込みレベルが高く、固定の間隔で起動します。しかし、profile-*n* プローブのように各 CPU で起動するわけではありません。tick-*n* プローブは、毎回 1 個の CPU 上で起動します。実際の CPU は、時間の経過とともに変化します。profile-*n* プローブの場合と同じように、*n* はデフォルトでは秒当たりのレートですが、時間を表す接尾辞 (オプション) を付けることもできます。tick-*n* プローブには、定期的な出力、定期的なアクションの実行など、いくつかの用途があります。

引数

profile プローブの引数は、次の表のとおりです。

arg0	プローブが起動したときのカーネル内のプログラムカウンタ (PC)。プローブが起動したとき、カーネル内で現在のプロセスが実行されていなかった場合はゼロになります。
arg1	プローブが起動したときのユーザーレベルのプログラムカウンタ (PC)。プローブが起動したとき、カーネル内で現在のプロセスが実行されていた場合はゼロになります。

この説明からわかるように、arg0 がゼロ以外の値の場合 arg1 はゼロ、arg0 がゼロの場合 arg1 はゼロ以外になります。この特性を利用すると、arg0 と arg1 を使ってユーザーレベルとカーネルレベルを区別できます。以下に例を示します。

```
profile-1ms
{
    @ticks[arg0 ? "kernel" : "user"] = count();
}
```

タイマー分解能

profile プロバイダは、オペレーティングシステム内の任意の分解能のタイマーを使用します。任意の分解時間ベースの割り込みを完全にサポートしないアーキテクチャでは、周波数が、カーネル変数 hz で指定されたシステムクロックのクロック周波数によって制限されます。こうしたアーキテクチャーでは、hz より周波数が高いプローブは、1/hz 秒ごとに何回か起動します。たとえば、こうしたアーキテクチャで hz が 100 に設定されている場合、1000 ヘルツの profile プローブは、10 ミリ秒ごとに連続してすばやく 10 回起動します。任意の分解能をサポートするプラットフォームでは、1000 ヘルツの profile プローブは、ミリ秒ごとに 1 回ずつ起動します。

以下は、アーキテクチャの分解能をテストする例です。

```
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);
}
```

任意の分解能の profile プローブをサポートするアーキテクチャでは、このスクリプトを実行すると、 結果は均一の分布になります。

```
# 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
```

任意の分解能の `profile` プローブをサポートしないアーキテクチャでは、このスクリプトを実行すると、 結果は均一の分布になりません。

```
# 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
```

こうしたアーキテクチャで、実効プロファイルの分解能を改善するには、`/etc/system` ファイルを手作業で編集して、`hz` の値をチューニングします。

現在、すべての UltraSPARC (`sun4u`) は、任意の分解能の `profile` プローブをサポートしています。多くの x86 アーキテクチャ (`i86pc`) も、任意の分解能の `profile` プローブをサポートしていますが、バージョンの古いものの中には、一部サポートしないものもあります。

プローブの作成

`profile` プロバイダは、ほかのプロバイダとは違って、必要に応じて動的にプローブを作成します。`dtrace -l -P profile` などを実行したとき、表示される全プローブのリストに必要なプロファイルプローブが含まれていない場合がありますが、このプローブは、明示的に有効にすると作成されます。

任意の分解能の `profile` プローブをサポートするアーキテクチャでは、時間間隔が短すぎると、マシンは時間ベースの割り込みの処理に追われることになります。その結果、マシンの本来のサービスが妨害されます。この問題を防ぐため、`profile` プロバイダは、200 マイクロ秒未満の間隔になるプローブの作成を拒否します。このとき、メッセージは表示されません。

安定性

以下の表に、profile プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	共通
モジュール	変更の可能性あり	変更の可能性あり	不明
機能	非公開	非公開	不明
名前	発展中	発展中	共通
引数	発展中	発展中	共通

◆ ◆ ◆ 第 20 章

20 fbt プロバイダ

この章では、関数境界トレース (Function Boundary Tracing、FBT) プロバイダについて説明します。このプロバイダは、Solaris カーネルのほとんどの関数の開始 (entry) と終了 (return) に関連したプローブを提供します。関数は、プログラムテキストの基本単位です。上手に設計されたシステムでは、各関数が、指定された 1 つのオブジェクトまたは類似した複数のオブジェクトに対して、明確に定義された操作を個別に実行します。このため、どんなに小規模な Solaris システムであっても、FBT は、20,000 個程度のプローブを提供します。

ほかの DTrace プロバイダ同様、FBT でも、明示的に有効にしない限り、プローブの及ぼす影響は皆無です。FBT プローブを有効にすると、それに関連付けられた関数だけがプローブの影響を受けます。FBT 実装は、命令セットアーキテクチャに固有のものですが、FBT は、SPARC と x86 の両方のプラットフォームに実装されています。命令セットごとに、少数ですが、*FBT* では計測できない関数があります。この関数は、「リーフ関数」と呼ばれています。リーフ関数はコンパイラによって高度に最適化されており、別の関数を呼び出しません。リーフ関数のプローブは、DTrace 内には存在しません。

FBT プローブを効果的に利用するためには、オペレーティングシステム実装に関する知識が必要です。したがって、FBT は、カーネルソフトウェアの開発時や、ほかのプロバイダでは十分でない場合にのみ使用することをお勧めします。`syscall`、`sched`、`proc`、`io` などのその他の DTrace プロバイダは、オペレーティングシステム実装の知識がなくても使用できます。これらのプロバイダを使って、システム分析に関するほとんどの問題の答えを引き出すことができます。

プローブ

FBT は、カーネル内のほとんどの関数の「境界」でプローブを提供します。関数の境界を越えるのは、関数に入るときと関数から抜けるときです。FBT は、カーネル内の各関数に、関数に入るときと関数から抜けるときに 1 つずつ、合計 2 つの関数を提供しています。これらのプローブにはそれぞれ、`entry`、`return` という名前が付けられています。関数名とモジュール名は、プローブの一部として指定されます。すべての FBT プローブは、関数名とモジュール名を指定します。

プローブ引数

entry プローブ

`entry` プローブの引数は、対応するオペレーティングシステムカーネル関数の引数と同じです。これらの引数にアクセスするときは、`args[]` 配列を使って、引数の型を想定してアクセスします。これらの引数は、次の変数を使うことで、`int64_t` 型としてアクセスできます。`arg0` .. `arg`*n* の変数です。

return プローブ

エントリポイント 1 つに対して、呼び出し元へ戻るポイントを複数持っている関数もあります。通常、ユーザーにとって重要なのは、関数から返される値か、関数が終了したという事実かであって、具体的なリターンパスではありません。このため、FBT は、関数の復帰 (複数存在) を単一の `return` プローブで収集します。正確なリターンパスが必要な場合は、`return` プローブの `args[0]` の値を確認します。この値は、関数テキスト内の復帰命令のオフセット (バイト単位) を表しています。

関数に戻り値がある場合、この戻り値は `args[1]` に格納されます。関数に戻り値がない場合、`args[1]` は定義されません。

例

FBT を使用すると、カーネルの実装を簡単に調べることができます。以下に、`xclock` プロセスの最初の `ioctl(2)` を記録し、カーネル経由で続きのコードパスをたどるスクリプトの例を示します。

```
/*
 * 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);
}
```

このスクリプトを実行すると、次のような出力が得られます。

```
# 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
```

上記の出力では、xclock プロセスが、ソケットと関連付けられているように見えるファイル記述子で ioctl() を呼び出しています。

FBT で、カーネルドライバに関する情報を得ることもできます。たとえば、ssd(7D) ドライバには、EIO を返すコードパスが多数あります。FBT を使用すると、エラーを出した正確なコードパスを簡単に突き止めることができます。次の例を参照してください。

```
fbt:ssd::return
/arg1 == EIO/
{
    printf("%s+%x returned EIO.", probefunc, arg0);
}
```

EIO の詳細情報を得たい場合は、すべての fbt プローブを投機的にトレースし、関数の戻り値に基づいて commit() (または discard()) を実行します。投機トレースについては、第 13 章「投機トレース」を参照してください。

FBT を使って、指定のモジュール内で呼び出された関数の情報を得ることもできます。以下は、UFS で呼び出されたすべての関数を一覧表示する例です。

```
# 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
```

カーネル関数の引数の目的を理解している場合は、FBT を使って、この関数を呼び出す理由やその方法を確認できます。たとえば、putnext(9F) の最初のメンバーは、queue(9S) 構造体のポインタになります。queue 構造体の q_qinfo メンバーは、qinit(9S) 構造体のポインタになります。qinit 構造体の qi_minfo メンバーは、module_info(9S) 構造体のポインタを持っています。module_info 構造体の mi_idname メンバーには、モジュール名が格納されます。以下の例では、putnext 内で FBT プローブを使用することによってこの情報をまとめ、モジュール名から putnext(9F) 呼び出しを追跡します。

```
fbt::putnext:entry
{
```

```
    @calls[stringof(args[0]->q_qinfo->qi_minfo->mi_idname)] = count();
}
```

このスクリプトを実行すると、次のような出力が得られます。

```
# 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
```

FBT を使って、特定の関数で消費された時間を調べることもできます。以下の例では、DDI 遅延ルーチン drv_usecwait(9F) と delay(9F) の呼び出し元を特定する方法を示します。

```
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;
}
```

このスクリプトは、ブート中に実行されることに意味があります。システムブート中の匿名トレースの実行手順については、第 36 章「匿名トレース」で説明します。リブート時に、次のような出力が得られます。

```
# dtrace -ae

              ataʻata_wait+0x34
              ataʻata_id_common+0xf5
              ataʻata_disk_id+0x20
              ataʻata_drive_type+0x9a
              ataʻata_init_drive+0xa2
              ataʻata_attach+0x50
              genunixʻdevi_attach+0x75
              genunixʻattach_node+0xb2
              genunixʻi_ndi_config_node+0x97
              genunixʻi_ddi_attachchild+0x4b
              genunixʻdevi_attach_node+0x3d
              genunixʻdevi_config_one+0x1d0
              genunixʻndi_devi_config_one+0xb0
              devfsʻdv_find+0x125
              devfsʻdevfs_lookup+0x40
              genunixʻfop_lookup+0x21
              genunixʻlookuppnvp+0x236
              genunixʻlookuppnat+0xe7
              genunixʻlookupnameat+0x87
              genunixʻcstatat_getvp+0x134

            value  ------------- Distribution ------------- count
             2048 |                                         0
             4096 |@@@@@@@@@@@@@@@@@@@@@@@@                 4105
             8192 |@@@@                                     783
            16384 |@@@@@@@@@@@@@@@                          2793
            32768 |                                         16
            65536 |                                         0


              kb8042ʻkb8042_wait_poweron+0x29
              kb8042ʻkb8042_init+0x22
              kb8042ʻkb8042_attach+0xd6
              genunixʻdevi_attach+0x75
              genunixʻattach_node+0xb2
              genunixʻi_ndi_config_node+0x97
              genunixʻi_ddi_attachchild+0x4b
              genunixʻdevi_attach_node+0x3d
              genunixʻdevi_config_one+0x1d0
              genunixʻndi_devi_config_one+0xb0
              genunixʻresolve_pathname+0xa5
              genunixʻddi_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
```

末尾呼び出しの最適化

ある関数が別の関数を呼び出して終了したとき、コンパイラは、「末尾呼び出しの最適化」を行うことができます。その結果、呼び出し元のスタックフレームを呼び出された関数で再利用できるようになります。この手続きは、SPARC アーキテクチャでよく行われます。SPARC アーキテクチャのコンパイラは、レジスタウィンドウの負荷を極力抑えるため、呼び出される側の関数で呼び出し元のレジスタウィンドウを再利用します。

この最適化により、呼び出し元関数の `return` プローブは、呼び出される側の `entry` プローブより先に起動するようになります。この順序は、かなり混乱を招きやすくなっています。たとえば、特定の関数から呼び出されたすべての関数と、この特定の関数が呼び出すすべての関数を記録する場合、次のようなスクリプトを使用します。

```
fbt::foo:entry
{
    self->traceme = 1;
}

fbt:::entry
/self->traceme/
{
    printf("called %s", probefunc);
}

fbt::foo:return
/self->traceme/
{
    self->traceme = 0;
}
```

しかし、`foo()` が最適化された末尾呼び出しで終わる場合、末尾で呼び出された関数と、この関数によって呼び出された関数は捕捉されません。動的にカーネルの最適化を解除することはできません。また、DTrace は、コードの構造を偽ることを望みません。このため、末尾呼び出しの最適化がいつ行われるかを意識する必要があります。

末尾呼び出しの最適化は、通常、次の例のようなソースコードで行われます。

```
    return (bar());
```

次のようなソースコードで行われる場合もあります。

```
    (void) bar();
    return;
```

逆に、次の例のような終わり方の関数ソースコードでは、`bar()` の呼び出しを最適化することができません。これは、`bar()` の呼び出しが末尾呼び出しではないためです。

```
    bar();
    return (rval);
```

末尾呼び出しの最適化が行われているかどうかは、次のようにして判断できます。

- DTrace の実行中に、問題の `return` プローブの `arg0` をトレースします。`arg0` には、関数内の復帰命令のオフセットが格納されます。
- DTrace の停止後、`mdb(1)` を使って関数を調べます。トレースされたオフセットに、この関数からの復帰命令ではなく、別の関数の呼び出しが含まれている場合、末尾呼び出しの最適化が行われています。

命令セットのアーキテクチャ上の理由から、末尾呼び出しの最適化は、x86 システムよりも SPARC システムでよく使用されます。以下は、mdb を使って、カーネルの dup() 関数内で末尾呼び出しの最適化を検出する例です。

```
# dtrace -q -n fbt::dup:return'{printf("%s+0x%x", probefunc, arg0);}'
```

このコマンドの実行中に、bash プロセスなど、dup(2) を実行するプログラムを実行します。このコマンドからは、次のような出力が得られます。

```
dup+0x10
^C
```

mdb を使って関数を調べましょう。

```
# 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
```

dup+0x10 が fcntl() 関数の呼び出しであり、ret 命令でないことが、この出力からわかります。したがって、fcntl() の呼び出しは、末尾呼び出しの最適化の一例になっています。

アセンブリ関数

関数に入るだけで抜けない場合や、関数に入らずに抜けるだけの場合が存在します。まれにあるこのような関数は、概して、ハンドコードされたアセンブリルーチンで、ほかのハンドコードされたアセンブリ関数の中へと分岐しています。これらの関数が、解析を妨げることがあってはなりません。 分岐先の関数は、分岐元の関数の呼び出し元へ復帰する必要があります。つまり、すべての FBT プローブを有効にした場合、ある関数へ入る動作と別の関数から復帰する動作が同じスタック深度で行われるべきです。

命令セットの制限

一部の関数は、FBT で計測できません。計測不能な関数は、命令セットアーキテクチャに固有の関数です。

x86 の制限

x86 システム上でスタックフレームを作成しない関数は、FBT で計測できません。x86 のレジスタセットは非常に小さいので、ほとんどの関数は、データをスタックに格納するため、スタックフレームを作成します。しかし、一部の x86 関数はスタックフレームを作成しないため、計測できません。x86 プラットフォーム上で計測できない関数の数は決まっていませんが、通常は全体の 5 % 未満です。

SPARC の制限

アセンブリ言語で SPARC システムにハンドコードされたリーフルーチンは、FBT では計測できません。カーネルの大部分は C で書かれているので、C で書かれた関数はすべて FBT で計測できます。

ブレークポイントとの相互作用

FBT は、カーネルテキストを動的に変更することによって機能します。カーネルのブレークポイントも、カーネルテキストを変更することによって機能します。このため、カーネルのブレークポイントを DTrace のロード前の開始時 (entry) または終了時 (return) に配置した場合、FBT はこの関数にプローブを提供しなくなります。これは、その後カーネルのブレークポイントを削除したとしても変わりません。カーネルのブレークポイントを DTrace のロード後に配置した場合、カーネルのブレークポイントと DTrace プローブの両方が、テキスト内の同じポイントに対応するようになります。この場合、デバッガがカーネルを再開すると、まずブレークポイントがトリガーされ、次にプローブが起動します。カーネルのブレークポイントと DTrace は、なるべく併用しないでください。ブレークポイントが必要な場合は、DTrace の `breakpoint ()` アクションを使用してください。

モジュールのロード

Solaris カーネルは、カーネルモジュールを動的にロードしたり、アンロードしたりできます。FBT がロードされ、モジュールが動的にロードされると、FBT により、新しいモジュールに関連付けられた新しいプローブが自動的に提供されます。ロードされたモジュールの FBT プローブが有効化されていない場合、このモジュールの読み込みは解除されます。さらに、このアンロードに伴って、対応するプローブが破棄されます。ロードされたモジュールの FBT プローブが有効である場合、モジュールはビジー状態と見なされるので、アンロードできません。

安定性

以下の表に、FBT プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	非公開	非公開	ISA

FBT はカーネルの実装を公開するので、「安定」に該当するものはありません。モジュールと関数では、名前の安定性およびデータの安定性が「非公開」です。プロバイダと名前のデータの安定性は「発展中」になっていますが、その他のすべてのデータの安定性は「非公開」になっています。 これらは、現在の実装に影響を受けています。FBT の依存クラスは ISA です。FBT は現在の命令セットアーキテクチャ全部で使用できますが、将来の任意の命令セットアーキテクチャで FBT を使用できる保証はありません。

◆ ◆ ◆ 第 21 章

21 syscall プロバイダ

syscall プロバイダは、システム内のすべてのシステムコールに開始時 (entry) と終了時 (return) のプローブを提供します。システムコールは、ユーザーレベルのアプリケーションとオペレーティングシステムカーネルを結びつける主要なインタフェースです。このため、syscall プロバイダは、システム関連のアプリケーションの動作について非常に多くの情報を提供します。

プローブ

syscall プロバイダは、システムコールごとにプローブを 1 組ずつ提供します。 うち 1 つは、システムコールに入る前に起動する entry プローブです。もう 1 つは、システムコールが完了したあと、ユーザーレベルの制御に戻る前に起動する return プローブです。どの syscall プローブでも、関数名は計測されるシステムコールの名前です。モジュール名は未定義です。

syscall プロバイダが提供するシステムコールの名前は、/etc/name_to_sysnum ファイルで確認できます。多くの場合、syscall が提供するシステムコールの名前は、マニュアルページのセクション 2 の名前と対応しています。しかし、syscall プロバイダが提供するプローブの中には、文書化されたシステムコールに直接対応していないものもあります。以下では、この矛盾の主な原因について説明します。

古いシステムコール

syscall プロバイダが提供するシステムコールの名前が、以前の実装内容を反映している場合があります。たとえば、以前の UNIX™ を反映して、/etc/name_to_sysnum ファイル内では exit(2) の名前が rexit になっています。同様に、time(2) の名前は gtime、execle(2) と execve(2) の名前はどちらも exece になっています。

サブコード化されたシステムコール

セクション 2 のシステムコールの一部は、文書化されていないシステムコールの下位操作として実装されています。たとえば、System V セマフォ関連のシステムコール (semctl(2)、semget(2)、semids(2)、semop(2)、および semtimedop(2)) は、semsys という単一のシステムコールの下位操作として実装されています。semsys システムコールは、最初の引数として、要求されたシステムコールを示す実装固有の「サブコード」(SEMCTL、SEMGET、SEMIDS、SEMOP、または SEMTIMEDOP) を取ります。単一のシステムコールを使用して複数のシステムコールを実装しているため、複数の System V セマフォに対して syscall プローブは syscall::semsys:entry および syscall::semsys:return の 1 組だけ存在します。

大規模ファイルのシステムコール

4G バイトを超える大規模ファイルをサポートする 32 ビットプログラムは、64 ビットファイルのオフセットを処理できなければなりません。大規模ファイルの場合、大規模オフセットを使用するため、大規模ファイルを操作するときは、複数のシステムインタフェースを並行して利用します (lf64(5) のマニュアルページを参照)。これらのインタフェースについては、lf64 に記載されています。ただし、インタフェースごとに個別のマニュアルページは用意されていません。これらの大規模ファイルシステムコールインタフェースは、表 21–1 のように、独自の syscall プローブとして記載されています。

表 21–1　syscall 大規模ファイルプローブ

大規模ファイル syscall プローブ	システムコール
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)

表 21–1 sycall 大規模ファイルプローブ (続き)

大規模ファイル syscall プローブ	システムコール
stat64	stat(2)
statvfs64	statvfs(2)

非公開システムコール

一部のシステムコールは、ユーザーとカーネル間の境界にまたがる Solaris サブシステムの非公開実装です。このため、マニュアルページのセクション2 に、これらのシステムコールはありませんこのようなシステムコールの例として、POSIX.4 メッセージキューの実装の一部として使用される signotify システムコールや、fuser(1M) を実装するために使用される utssys システムコールなどがあります。

引数

entry プローブの場合、引数 (arg0 .. arg*n*) はシステムコールの引数です。return プローブの場合、arg0 と arg1 の両方に戻り値が格納されます。システムコールに失敗した場合は、D 変数 errno にゼロ以外の値が入ります。

安定性

以下の表に、syscall プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	共通
モジュール	非公開	非公開	不明
機能	変更の可能性あり	変更の可能性あり	ISA
名前	発展中	発展中	共通
引数	変更の可能性あり	変更の可能性あり	ISA

◆ ◆ ◆ 第 22 章

22 sdt プロバイダ

静的定義トレース (Statically Defined Tracing、SDT) プロバイダは、ソフトウェアプログラマが正式に指定した位置でプローブを作成します。プログラマは、SDT 機構を利用して、DTrace ユーザーにとって重要な箇所をプローブの設定位置として意識的に選択できます。さらに、このプローブに、その位置についてよくわかるようなプローブ名を付けることができます。Solaris カーネルにも少数の SDT プローブが定義されており、将来はさらに追加される予定です。DTrace はまた、ユーザーアプリケーション開発者を対象に、静的プローブを定義する機構を提供しています。これについては、第 34 章「ユーザーアプリケーション向けの静的に定義されたトレース」で説明します。

プローブ

表 22–1 は、Solaris カーネルに定義されている SDT プローブです。これらのプローブの名前の安定性とデータの安定性は、どちらも「非公開」です。これは、ここでの記述がカーネルの実装を反映しているため、確約されたインタフェースとして推測されるべきでないからです。DTrace の安定性機構の詳細については、239 ページの「安定性」を参照してください。

表 22–1 SDT プローブ

プローブ名	説明	arg0
callout-start	コールアウトの実行直前に起動するプローブ (<sys/callo.h> を参照)。コールアウトは、timeout(9F) の実装を表すものであり、システムクロックによって定期的に実行されます。	実行されるコールアウトを表す callout_t のポインタ (<sys/callo.h> を参照)。

表 22–1 SDT プローブ (続き)

プローブ名	説明	arg0
callout-end	コールアウトの実行直後に起動するプローブ (<sys/callo.h> を参照)。	直前に実行されたコールアウトを表す callout_t のポインタ (<sys/callo.h> を参照)。
interrupt-start	デバイスの割り込みハンドラを呼び出す直前に起動するプローブ。	割り込みデバイスを表す dev_info 構造体のポインタ (<sys/ddi_impldefs.h> を参照)。
interrupt-complete	デバイスの割り込みハンドラから復帰した直後に起動するプローブ。	割り込みデバイスを表す dev_info 構造体のポインタ (<sys/ddi_impldefs.h> を参照)。

例

以下は、1 秒に 1 回のペースでコールアウトの動作を調べるスクリプトです。

```
#pragma D option quiet

sdt:::callout-start
{
    @callouts[((callout_t *)arg0)->c_func] = count();
}

tick-1sec
{
    printa("%40a %10@d\n", @callouts);
    clear(@callouts);
}
```

この例を実行すると、そのシステム内で timeout(9F) をよく使用するユーザーがわかります。次の出力例を参照してください。

```
# 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
```

timeout(9F) インタフェースは、単一のタイマーの有効期限を出力するだけです。timeout() のインターバルタイマー機能を利用する場合、通常 timeout() ハンドラから timeout を再インストールします。以下に例を示します。

```
#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);
}
```

このスクリプトを実行し、しばらく待ってから Control-C キーを押すと、次のような出力が得られます。

```
# 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
```

この出力からわかるように、uhci(7D) ドライバ内の `uhci_handle_root_hub_status_change()` は、このシステム上のもっとも短い間隔で呼び出されるインターバルタイマー (システムクロック刻みに合わせて呼び出される) です。

割り込みアクティビティに関する情報は、`interrupt-start` プローブから得ることができます。次の例では、ドライバ名を指定して、割り込みハンドラの実行にかかった時間を調べています。

```
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);
}
```

このスクリプトを実行すると、次のような出力が得られます。

```
# 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
```

SDT プローブの作成

デバイスドライバの開発者は、Solaris ドライバ内に独自の SDT プローブを作成したいと思うことがあるかもしれません。SDT プローブが無効にされている状態は、無操作マシン命令がいくつかある状態と実質的に同じです。したがって、必要に応じて SDT プローブをデバイスドライバに追加してかまいません。これらのプローブがパフォーマンスに悪影響を及ぼさないかぎり、出荷コード内に残しておいてもかまいません。

プローブの宣言

SDT プローブを宣言するときは、<sys/sdt.h> のマクロ DTRACE_PROBE 、DTRACE_PROBE1、DTRACE_PROBE2、 DTRACE_PROBE3、DTRACE_PROBE4 を使用します。SDT ベースのプローブのモジュール名は、カーネルモジュール名を反映しています。また、関数名は、そのプローブの関数を表しています。プローブ名は、DTRACE_PROBE*n* マクロで指定された名前によって決まります。この名前に 2 つの連続する下線 (__) が含まれていない場合、マクロに指定されたとおりのプローブ名になります。この名前に 2 つの連続する下線が含まれている場合、プローブ名では、この下線部分がダッシュ 1 個 (-) に変換されます。たとえば、DTRACE_PROBE マクロに transaction__start と指定されている場合、SDT プローブ名は transaction-start になります。このような置き換えが行われるので、C コード内のマクロ名が有効な C 識別子でなくても、文字列を指定しないで使用できます。

カーネルモジュール名と関数名は、DTrace によって、プローブを識別する組に含められます。したがって、名前空間の衝突を避けるためにプローブ名にこの情報を指定する必要性は特にありません。インストール済みのプローブと、DTrace ユーザーが確認できるフルネームを一覧するには、ドライバ上で dtrace -l -P sdt -m *module* (*module* はドライバ) を実行します。

プローブ引数

各 SDT プローブの引数は、対応する DTRACE_PROBE*n* マクロ参照に指定された引数になります。引数の数は、どのマクロを使ってプローブを作成したかによって異なります。 たとえば、DTRACE_PROBE1 は引数を 1 つ、DTRACE_PROBE2 は 2 つ (以下同様) 指定します。SDT プローブを宣言するときは、ポインタを間接参照せず、プローブ引数内の大域変数からロードしないようにすれば、無効時のプローブの影響を最小限に抑えることができます。ポインタの間接参照も、大域変数のロードも、D のプローブ有効化アクション内で安全に実行できます。こうしたアクションが必要なときには要求してかまいません。

安定性

以下の表に、SDT プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	非公開	非公開	ISA
引数	非公開	非公開	ISA

◆ ◆ ◆ 第 23 章

23 sysinfo プロバイダ

sysinfo プロバイダは、sys という名前で分類されるカーネル統計情報のプローブを使用できるようにします。これらの統計情報は、mpstat(1M) などのシステム監視ユーティリティーへの入力を提供するので、sysinfo プロバイダは、観測された異常動作の迅速な検査を可能にします。

プローブ

sysinfo プロバイダは、sys に分類されるカーネル統計情報内のフィールドに対応したプローブを使用できるようにします。sysinfo から提供されたプローブは、対応する sys 値が増分される直前に起動します。以下は、kstat(1M) コマンドを使って、sys に分類されるカーネル統計情報の名前と現在の値を両方表示する例です。

```
$ kstat -n sys
module: cpu                             instance: 0
name:   sys                             class:    misc
    bawrite                         123
    bread                           2899
    bwrite                          17995
...
```

表 23–1 に、sysinfo プローブを一覧します。

表 23–1 sysinfo プローブ

bawrite	バッファーからデバイスへの非同期書き出しが行われる直前に起動するプローブ。
bread	デバイスからのバッファーの物理読み取りが行われたときに起動するプローブ。bread は、デバイスがバッファーを要求したあと、処理の完了が保留される前に起動します。

表 23–1 sysinfo プローブ (続き)

bwrite	バッファーからデバイスへの書き出しが行われる直前に起動するプローブ。書き出しは、同期書き出し、非同期書き出しの両方を含みます。
idlethread	CPU がアイドルループに入ったときに起動するプローブ。
intrblk	割り込みスレッドがブロックされたときに起動するプローブ。
inv_swtch	実行中のスレッドが CPU の解放を強制されたとき起動するプローブ。
lread	デバイスからのバッファーの論理読み取りが行われたときに起動するプローブ。
lwrite	バッファーからデバイスへの論理書き込みが行われたときに起動するプローブ。
modload	カーネルモジュールがロードされたときに起動するプローブ。
modunload	カーネルモジュールがアンロードされたときに起動するプローブ。
msg	msgsnd(2) または msgrcv(2) システムコールが発行されたあと、メッセージキュー操作が行われる前に起動するプローブ。
mutex_adenters	所有されている適応型ロックの獲得が試みられたときに起動するプローブ。このプローブが起動するときは、lockstat プロバイダの adaptive-block プローブ、または adaptive-spin プローブも起動します。詳細については、第 18 章「lockstat プロバイダ」を参照してください。
namei	ファイルシステム内で名前の検索が試みられたときに起動するプローブ。
nthreads	スレッドが作成されたときに起動するプローブ。
phread	生の入出力読み取りが行われる直前に起動するプローブ。
phwrite	生の入出力書き込みが行われる直前に起動するプローブ。
procovf	システムのプロセステーブルのエントリがなくなったため新しいプロセスを作成できないときに起動するプローブ。
pswitch	CPU が実行スレッドを切り替えたときに起動するプローブ。
readch	正常に読み取りが行われたあと、この読み取りの実行スレッドに制御が移る前に起動するプローブ。読み取りに使用されるシステムコールは、read(2)、readv(2)、pread(2) のいずれかです。arg0 には、正常に読み取られたバイト数が格納されます。
rw_rdfails	書き込み側が読み取り/書き込みロックを保持している場合、または必要としている場合に、この読み取り/書き込みロックの読み取りロックが試みられたとき起動するプローブ。このプローブが起動するときは、lockstat プロバイダの rw-block プローブも起動します。詳細については、第 18 章「lockstat プロバイダ」を参照してください。

表 23–1 sysinfo プローブ (続き)

rw_wrfails	読み取り/書き込みロックが 1 つ以上の読み取り側、または単一の書き込み側によって保持されている場合に、この読み取り/書き込みロックを保持していない書き込み側による書き込みロックが試みられたとき起動するプローブ。このプローブが起動するときは、lockstat プロバイダの rw-block プローブも起動します。詳細については、第 18 章「lockstat プロバイダ」を参照してください。
sema	システムコール semop(2) が発行されたあと、セマフォー操作が行われる前に起動するプローブ。
sysexec	システムコール exec(2) が発行されたときに起動するプローブ。
sysfork	システムコール fork(2) が発行されたときに起動するプローブ。
sysread	システムコール read(2)、readv(2)、pread(2) のいずれかが呼び出されたときに起動するプローブ。
sysvfork	システムコール vfork(2) が発行されたときに起動するプローブ。
syswrite	システムコール write(2)、writev(2)、または pwrite(2) が発行されたときに起動するプローブ。
trap	プロセッサトラップが発生したときに起動するプローブ。ただし、一部のプロセッサ、特に UltraSPARC 系プロセッサでは、一部の軽量トラップを処理するときにこのプローブを起動しないことがあります。
ufsdirblk	UFS ファイルシステムによるディレクトリブロックの読み取りが行われたときに起動するプローブ。UFS については、ufs(7FS) のマニュアルページを参照してください。
ufsiget	i ノードの取得時に起動するプローブ。UFS については、ufs(7FS) のマニュアルページを参照してください。
ufsinopage	データページが関連付けられていないコア内の i ノードの再利用が可能になったあと起動するプローブ。UFS については、ufs(7FS) のマニュアルページを参照してください。
ufsipage	データページが関連付けられたコア内の i ノードの再利用が可能になったあと起動するプローブ。このプローブは、関連付けられたデータページがディスクにフラッシュされたあとで起動します。UFS については、ufs(7FS) のマニュアルページを参照してください。
writech	正常に書き込みが行われたあと、この書き込みの実行スレッドに制御が移る前に起動するプローブ。書き込みに使用されるシステムコールは、write(2)、writev(2)、pwrite(2) のいずれかです。arg0 には、正常に書き込まれたバイト数が格納されます。
xcalls	クロスコールが行われる直前に起動するプローブ。クロスコールは、一方の CPU から、別の CPU によるすばやい処理を要求するオペレーティングシステム機構です。

引数

sysinfo プローブには、次の引数があります。

arg0	統計情報の増分値。この引数の値は、ほとんどのプローブでは 1 ですが、別の値をとるプローブもあります。
arg1	増分対象の統計情報の、現在値のポインタ。arg0 の値に従って増分される量を 64 ビット値で表します。このポインタを間接参照することにより、コンシューマ側で、プローブの統計情報の現在のカウントを特定できます。
arg2	統計情報の増分が行われる CPU の cpu_t 構造体のポインタ。この構造体は <sys/cpuvar.h> に定義されていますが、カーネル実装の一部であるため、「非公開」と見なします。

ほとんどの sysinfo プローブでは、arg0 の値は 1 です。ただし、readch プローブでは arg0 が読み取られたバイト数に、writech プローブでは書き込まれたバイト数になります。この機能を利用すると、実行可能ファイルの名前から読み取りのサイズを特定できます。次の例を参照してください。

```
# 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
```

sysinfo プロバイダは、arg2 を、カーネル実装の内部構造体である cpu_t のポインタに設定します。sysinfo プローブは、統計情報の増分が行われる CPU 上で起動します。情報を得たい CPU を特定するには、cpu_t 構造体の cpu_id メンバーを使用します。

例

以下は、mpstat(1M) の出力例です。

```
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
```

この出力をよく見ると、システムが比較的アイドル状態にあるのに xcal フィールドの値が大きすぎることがわかります。mpstat は、xcal フィールドの値を決めるとき、sys カーネル統計情報の xcalls フィールドを調べます。したがって、次の例のように xcalls sysinfo プローブを有効にすれば、この異常について簡単に調べることができます。

```
# 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
```

この出力から、クロスコールのソースをどこで探せばよいかがわかります。ほとんどのクロスコールは、find(1) プロセス数個に影響を受けています。問題をもっと詳しく調べるには、次の D スクリプトを使用します。

```
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;
}
```

このスクリプトでは、syscall プロバイダを使って、find からのクロスコールがどのシステムコールに起因するか調べます。ページフォルトに起因するクロスコールなど、システムコールに起因しないクロスコールも存在します。この場合は <none> と出力されます。このスクリプトを実行すると、次のような出力が得られます。

```
# dtrace -s ./find.d
 dtrace: script ’./find.d’ matched 444 probes
^C
  <none>                                                          2
  lstat64                                                      2433
  getdents64                                                  14873
```

find によるクロスコールの大部分が、システムコール getdents(2) によって行われています。以降の作業は、調べたい内容に応じて異なります。たとえば、find プロセスが getdents を呼び出している理由を調べたい場合は、find がクロスコールを引き起こしたときに ustack() を集積するような D スクリプトを作成するとよいでしょう。getdents の呼び出しがクロスコールを引き起こす理由を調べたい場合は、find がクロスコールを引き起こしたときに stack() を集積するような D スクリプトを作成するとよいでしょう。次の作業が何であっても、xcalls プローブを利用することで、異常な監視結果が出力された原因を簡単に突き止めることができます。

安定性

以下の表に、sysinfo プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	非公開	非公開	ISA

◆ ◆ ◆ 第 24 章

24 vminfo プロバイダ

vminfo プロバイダは、vm カーネル統計情報のプローブを使用できるようにします。これらの統計情報は、vmstat(1M) などのシステム監視ユーティリティーへの入力を提供するので、vminfo プロバイダは、観測された異常動作の迅速な検査を可能にします。

プローブ

vminfo プロバイダは、vm に分類されるカーネル統計情報内のフィールドに対応したプローブを使用できるようにします。vminfo から提供されたプローブは、対応する vm 値が増分される直前に起動します。vm に分類されるカーネル統計情報の名前と現在の値を両方表示するには、以下の例に示すように、kstat(1M) コマンドを使用します。

```
$ 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
        ...
```

Table 23–1 に、表 24–1 プローブを一覧します。

表 24–1 vminfo プローブ

anonfree	ページングの一環として、変更されていない匿名ページが解放されたときに起動するプローブ。匿名ページとは、ファイルに関連したページを持たないページのことです。こうしたページは、ヒープメモリーやスタックメモリーのほか、zero(7D) の明示的なマッピングによって取得したメモリー内に存在します。
anonpgin	スワップデバイスから匿名ページがページインされたときに起動するプローブ。
anonpgout	変更された匿名ページがスワップデバイスへページアウトされたときに起動するプローブ。
as_fault	ページ上で、保護フォルト、書き込み時コピーフォルト以外のフォルトが発生したときに起動するプローブ。
cow_fault	ページ上で書き込み時コピーフォルトが発生したときに起動するプローブ。arg0 には、書き込み時コピーの結果作成されたページの数が格納されます。
dfree	ページングの結果ページが解放されたときに起動するプローブ。dfree が起動すると、anonfree、execfree、fsfree のいずれかが続いて起動します。
execfree	ページングの結果、変更されていない実行可能ページが解放されたときに起動するプローブ。
execpgin	バッキングストアから実行可能ページがページインされたときに起動するプローブ。
execpgout	変更された実行可能ページがバッキングストアへページアウトされたときに起動するプローブ。ほとんどの場合、実行可能ページのページングは execfree によって行われます。execpgout は、メモリー内で実行可能ページに変更が加えられたときだけ起動します。ただし、このような状況はめったに起こりません。
fsfree	ページングの一環として、変更されていないファイルシステムデータページが解放されたときに起動するプローブ。
fspgin	バッキングストアからファイルシステムページがページインされたときに起動するプローブ。
fspgout	変更されたファイルシステムページがバッキングストアへページアウトされたときに起動するプローブ。
kernel_asflt	ページ上でカーネルによるページフォルトが発生したとき、固有のアドレス空間で起動するプローブ。kernel_asflt の起動直前には必ず as_fault プローブが起動しているはずです。
maj_fault	ページフォルトの結果としてバッキングストアやスワップデバイスからの入出力が行われたときに起動するプローブ。maj_fault の起動直前には必ず pgin プローブが起動しているはずです。

表 24–1 vminfo プローブ (続き)

pgfrec	解放されたページのリストからページが再生されたときに起動するプローブ。
pgin	バッキングストアまたはスワップデバイスからページがページインされたときに起動するプローブ。このプローブは、maj_fault とは別のものです。maj_fault は、ページフォルトの結果としてページインが行われた場合にしか起動しません。これに対して、pgin は、理由はどうあれ、ページインが行われるたびに起動します。
pgout	バッキングストアまたはスワップデバイスへページがページアウトされたときに起動するプローブ。
pgpgin	バッキングストアまたはスワップデバイスからページがページインされたときに起動するプローブ。pgpgin と pgin の唯一の違いは、pgpgin では arg0 にページインされたページ数が格納される点です。pgin では arg0 に必ず 1 が格納されます。
pgpgout	バッキングストアまたはスワップデバイスへページがページアウトされたときに起動するプローブ。pgpgout と pgout の唯一の違いは、pgpgout では arg0 にページアウトされたページ数が格納される点です。pgout では arg0 に必ず 1 が格納されます。
pgrec	ページが再生されたときに起動するプローブ。
pgrrun	ページャがスケジュールされたときに起動するプローブ。
pgswapin	スワップアウトされたプロセスからページがスワップインされたときに起動するプローブ。スワップインされたページの数は、arg0 に格納されます。
pgswapout	プロセスのスワップアウトの一環としてページがスワップアウトされたときに起動するプローブ。スワップアウトされたページの数は、arg0 に格納されます。
prot_fault	保護違反によってページフォルトが発生したときに起動するプローブ。
rev	ページデーモンが全ページの巡回を新たに開始したときに起動するプローブ。
scan	ページデーモンがページを検査するときに起動するプローブ。
softlock	ページにソフトウェアロックを設定する作業でページフォルトが発生したときに起動するプローブ。
swapin	スワップアウトされたプロセスが再度スワップインされたときに起動するプローブ。
swapout	プロセスがスワップアウトされたときに起動するプローブ。
zfod	要求に応じてゼロに初期化されたページが作成されたときに起動するプローブ。

引数

arg0	統計情報の増分値。ほとんどのプローブでは、この引数の値は常に 1 です。ただし、表 24–1 に示すように、その他の値をとるプローブもあります。
arg1	増分対象の統計情報の、現在値のポインタ。arg0 の値に従って増分される量を 64 ビット値で表します。このポインタを間接参照することにより、コンシューマ側で、プローブの統計情報の現在のカウントを特定できます。

例

以下は、vmstat(1M) の出力例です。

```
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
```

pi 列には、ページインされたページ数が出力されています。vminfo プロバイダを使用すると、これらのページインのソースの詳細情報を得ることができます。次の例を参照してください。

```
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
```

この出力からわかるように、ほとんどのページインは、StarSuite™ ソフトウェア (soffice.bin) 関連の単一のプロセスによって行われています。soffice.bin の仮想メモリーの動作の様子を詳しく調べたいときは、すべての vminfo プローブを有効にします。以下は、StarSuite ソフトウェアの起動中に dtrace(1M) を実行する例です。

```
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
```

以下のスクリプト例では、StarSuite ソフトウェア 起動時の仮想メモリーの動作を詳しく調べることができます。

```
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);
}
```

今回も、StarSuiteソフトウェアの起動中にスクリプトを実行します。次に、新しい図形描画と新しいプレゼンテーションを作成し、すべてのファイルを閉じてアプリケーションを終了します。Dスクリプトを実行しているシェルウィンドウ内で、Control-Cキーを押します。結果として、時間の経過とともに変化する仮想メモリーの様子が出力されます。

```
# 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
```

仮想メモリーシステム関連の StarSuite の動作が出力されています。たとえば、`maj_fault` プローブは、アプリケーションを新しく起動するまで起動していません。期待したとおり、StarSuite のウォームスタートを行なっても、重大なフォルトは発生していません。`as_fault` の出力からは、アクティビティの最初のバーストから最後のバーストまでの様子がわかります。最初のバーストのあと、ユーザーがメニューを使って新しい図面を作成するまで遅延が発生します。その後、アイドル状態を経て、ユーザーが新しいプレゼンテーションをクリックした時点で最後のバーストが

起こります。zfod の出力からは、新しいプレゼンテーションの作成により、わずかな時間ですが、ゼロで初期化されたページにかなりの負荷がかかっていることがわかります。

以降の DTrace の調査は、調べる内容に応じて異なります。ゼロで初期化されたページの要求元について知りたい場合は、zfod の有効化の際に ustack() を集積できます。ゼロで初期化されたページのしきい値を設定し、しきい値を超えた場合に、stop() 破壊アクションを使って、悪影響を及ぼすプロセスを終了することもできます。この方法では、truss(1) や mdb(1) のような従来のデバッグツールも使用できます。vminfo プロバイダでは、vmstat(1M) などの従来のツールで出力された統計情報と、システム全体に影響を及ぼすような動作を引き起こすアプリケーションを関連付けることができます。

安定性

以下の表に、vminfo プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	非公開	非公開	ISA

◆ ◆ ◆ 第 25 章

25 proc プロバイダ

proc プロバイダは、プロセスの作成と終了、軽量プロセス (LWP) の作成と終了、新しいプログラムイメージの実行、およびシグナルの送信と処理に関連するプローブを使用できるようにします。

プローブ

表 25–1 では、proc プローブについて説明します。

表 25–1 proc プローブ

プローブ	説明
create	fork(2)、forkall(2)、fork1(2)、vfork(2) のいずれかを使ってプロセスが作成されたときに起動するプローブ。args[0] は、新しい子プロセスの psinfo_t をポイントしています。vfork とその他の fork のバリエーションとの違いは、フォークスレッドの lwpsinfo_t で、pr_flag メンバー内の PR_VFORKP を調べるとわかります。fork1 と forkall との違いは、親プロセスの psinfo_t (curpsinfo) と子プロセスの psinfo_t (args[0]) の両方の pr_nlwp メンバーを調べるとわかります。create プローブはプロセスが正常に作成されてから起動します。また、LWP はプロセスの作成時に作成されます。このため、まずプロセス作成時に作成された LWP に対して lwp-create が起動し、その後、新しいプロセスに対して create プローブが起動します。

表 25–1 proc プローブ (続き)

プローブ	説明
exec	exec(2) システムコールのバリエーション、exec(2)、execle(2)、execlp(2)、execv(2)、execve(2)、execvp(2) のいずれかを使って、プロセスが新しいプロセスイメージをロードするときに起動するプローブ。exec プローブは、プロセスイメージがロードされる前に起動します。したがって、execname、curpsinfo などのプロセス変数には、イメージがロードされる前のプロセス状態が格納されます。exec プローブの起動後しばらくすると、同じスレッド内で exec-failure プローブか exec-success プローブが起動します。args[0] は、新しいプロセスイメージのパスをポイントします。
exec-failure	exec(2) のバリエーションのいずれかが失敗したときに起動するプローブ。exec-failure プローブは、同じスレッド内で exec プローブが起動したあとでのみ起動します。args[0] には、errno(3C) の値が入ります。
exec-success	exec(2) のバリエーションのいずれかが成功したときに起動するプローブ。exec-success も、exec-failure プローブと同じように、同じスレッド内で exec プローブが起動したあとでのみ起動します。exec-success プローブが起動したときにはすでに、execname、curpsinfo などのプロセス変数には、新しいプロセスイメージがロードされたあとのプロセス状態が格納されています。
exit	現在のプロセスが終了する直前に起動するプローブ。終了理由を表す SIGCHLD siginfo.h(3HEAD) コードが、args[0] に格納されます。
fault	スレッドがマシンフォルトを検出したときに起動するプローブ。args[0] には、proc(4) に定義されているようにフォルトコードが格納されます。args[1] は、このフォルトの siginfo 構造体をポイントします。fault プローブをトリガーできるのは、シグナルを発行するフォルトだけです。
lwp-create	LWP の作成時に起動するプローブ。LWP は、通常、thr_create(3C) の結果として作成されます。args[0] は、新しいスレッドの lwpsinfo_t をポイントしています。args[1] は、このスレッドが含まれているプロセスの psinfo_t をポイントしています。
lwp-start	新しく作成された LWP のコンテキストで起動するプローブ。lwp-start プローブは、ユーザーレベルの命令が実行される前に起動します。LWP がプロセス内の最初の LWP である場合は、まず start プローブが起動し、続いて lwp-start プローブが起動します。
lwp-exit	シグナル、または thr_exit(3C) の明示的な呼び出しによって、LWP が終了する直前に起動するプローブ。

表 25–1 proc プローブ (続き)

プローブ	説明
signal-discard	シングルスレッドプロセスに送信されたシグナルが、プロセスによってブロックされず、かつ無視されたときに、起動するプローブ。この条件下では、シグナルは作成後すぐに破棄されます。ターゲットプロセスとスレッドの lwpsinfo_t と psinfo_t は、それぞれ args[0] と args[1] に格納されます。シグナル番号は args[2] に格納されます。
signal-send	スレッドまたはプロセスにシグナルが送信されたときに起動するプローブ。signal-send プローブは、送信側のプロセスとスレッドのコンテキストで起動します。受信側のプロセスとスレッドの lwpsinfo_t と psinfo_t は、それぞれ args[0] と args[1] に格納されます。シグナル番号は args[2] に格納されます。signal-send の起動後は、受信側のプロセスとスレッド内の signal-handle または signal-clear が必ず起動します。
signal-handle	スレッドがシグナルを処理する直前に起動するプローブ。signal-handle プローブは、シグナルを処理するスレッドのコンテキストで起動します。シグナル番号は args[0] に格納されます。シグナルの siginfo_t 構造体のポインタは、args[1] に格納されます。siginfo_t 構造体がない場合またはシグナルハンドラに SA_SIGINFO フラグが設定されていない場合は、args[1] の値は NULL です。プロセス内のシグナルハンドラのアドレスは、args[2] に格納されます。
signal-clear	ターゲットスレッドが sigwait(2)、sigwaitinfo(3RT)、sigtimedwait(3RT) のいずれかでシグナルを待機しているために、保留状態のシグナルが消去されるとき、起動するプローブ。この条件に当てはまる場合、保留状態のシグナルが消去され、呼び出し側にシグナル番号が返されます。シグナル番号は args[0] に格納されます。signal-clear は、待機していたスレッドのコンテキストで起動します。
start	新しく作成されたプロセスのコンテキストで起動するプローブ。start プローブは、プロセス内でユーザーレベルの命令が実行される前に起動します。

引数

表 25–2 に、proc プローブの引数の型を示します。引数については、表 25–1 を参照してください。

表 25–2 proc プローブ引数

プローブ	args[0]	args[1]	args[2]
create	psinfo_t *	—	—

表 25–2 proc プローブ引数 (続き)

プローブ	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

一部の proc プローブは、lwpsinfo_t 型の引数をとります。構造体 lwpsinfo_t については、proc(4) に文書化されています。DTrace コンシューマで使用可能な構造体 lwpsinfo_t の定義は、次のとおりです。

```
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;
```

pr_flag フィールドは、プロセスについて記述するフラグが格納されているビットマスクです。表 25–3 に、これらのフラグとその意味を示します。

表 25–3　pr_flag の値

PR_ISSYS	プロセスはシステムプロセスです。
PR_VFORKP	プロセスは vfork(2) の子の親です。
PR_FORK	プロセスがフォーク時継承モードを設定しました。
PR_RLC	プロセスが最終終了時実行モードを設定しました。
PR_KLC	プロセスが最終終了時終了モードを設定しました。
PR_ASYNC	プロセスが非同期停止モードを設定しました。
PR_MSACCT	プロセスが microstate アカウンティングを有効にしました。
PR_MSFORK	プロセスの microstate アカウンティングがフォーク時に継承されました。
PR_BPTADJ	プロセスがブレークポイント調整モードを設定しました。
PR_PTRACE	プロセスが ptrace(3C) 互換モードを設定しました。
PR_STOPPED	スレッドは停止した LWP です。
PR_ISTOP	スレッドは重要なイベントで停止した LWP です。
PR_DSTOP	スレッドは停止指令が有効になっている LWP です。
PR_STEP	スレッドはシングルステップ指令が有効になっている LWP です。
PR_ASLEEP	スレッドはシステムコール内で割り込み可能なスリープ状態になっている LWP です。
PR_DETACH	スレッドは切り離された LWP です。pthread_create(3C) と pthread_join(3C) のマニュアルページを参照してください。
PR_DAEMON	スレッドはデーモン LWP です。pthread_create(3C) のマニュアルページを参照してください。
PR_AGENT	スレッドはプロセスのエージェント LWP です。
PR_IDLE	スレッドは CPU のアイドルスレッドです。アイドルスレッドは、その CPU の実行キューが空のときにだけ実行されます。

pr_addr フィールドは、スレッドを表す非公開のカーネル内データ構造のアドレスです。データ構造体が非公開でも、pr_addr フィールドは、スレッドの有効期限が切れるまで、そのスレッド固有のトークンとして使用できます。

pr_wchan フィールドは、同期オブジェクト上でスレッドがスリープ状態になっているときに設定されます。pr_wchan フィールドの意味はカーネル実装以外に対しては公開されませんが、フィールド自体は、同期オブジェクトに固有のトークンとして使用できます。

pr_stype フィールドは、同期オブジェクト上でスレッドがスリープ状態になっているときに設定されます。表 25–4 に、pr_stype フィールドに入る値を一覧します。

表 25–4　pr_stype の値

SOBJ_MUTEX	カーネル相互排他同期オブジェクト。カーネル内の共有データ領域へのアクセスを直列化するために使用します。カーネル相互排他同期オブジェクトの詳細については、第 18 章「lockstat プロバイダ」と mutex_init(9F) のマニュアルページを参照してください。
SOBJ_RWLOCK	カーネル読み取り/書き込み同期オブジェクト。カーネル内の共有オブジェクトへのアクセスを同期化するために使用します。カーネル内の共有オブジェクトへのアクセスでは、同時に複数の読み取り、または単一の書き込みが許可されます。カーネル読み取り/書き込み同期オブジェクトの詳細については、第 18 章「lockstat プロバイダ」と rwlock(9F) のマニュアルページを参照してください。
SOBJ_CV	条件変数同期オブジェクト。条件変数は、一定の条件が揃うまで待機し続けるように設計されています。通常、条件変数は、共有データ領域へのアクセス以外の目的で同期化を行うために使用します。条件変数は、一般的にプロセスがプログラムに従って待機するときに使用されます。たとえば、poll(2)、pause(2)、wait(3C) などでブロックが発生した場合に使用されます。
SOBJ_SEMA	セマフォ同期オブジェクト。条件変数オブジェクトと同じく、所有権の概念を追跡しない汎用同期オブジェクトです。所有権は、Solaris カーネル内で優先順位の継承を行うときに必要になります。このため、セマフォオブジェクト内に固有の所有権がない場合、これらを広く利用することができません。詳細については、semaphore(9F)を参照してください。
SOBJ_USER	ユーザーレベル同期オブジェクト。ユーザーレベル同期オブジェクト上でのすべてのブロックは、SOBJ_USER 同期オブジェクトで処理されます。ユーザーレベル同期オブジェクトには、mutex_init(3C)、sema_init(3C)、rwlock_init(3C)、cond_init(3C) で作成されたものと、これらと同等の POSIX コマンドで作成されたものがあります。
SOBJ_USER_PI	優先順位の継承を行うユーザーレベル同期オブジェクト。所有権を追跡する一部のユーザーレベル同期オブジェクトでは、優先順位の継承も行うことができます。たとえば、pthread_mutex_init(3C) で作成された相互排他オブジェクトは、pthread_mutexattr_setprotocol(3C) を使って優先順位を継承できます。

表 25–4 pr_stype の値 (続き)

SOBJ_SHUTTLE	往復同期オブジェクト。往復オブジェクトは、door の実装に使用します。詳細については、door_create(3DOOR) を参照してください。

pr_state フィールドには、表 25–5 に示すいずれかの値が設定されます。pr_sname フィールドには、同じ表の丸括弧内の文字が設定されます。

表 25–5 pr_state の値

SSLEEP (S)	スレッドはスリープ中です。sched:::sleep プローブは、スレッドが SSLEEP 状態に移行する直前に起動します。
SRUN (R)	スレッドは実行可能ですが、現在実行されていません。sched:::enqueue プローブは、スレッドが SRUN 状態に移行する直前に起動します。
SZOMB (Z)	スレッドはゾンビ LWP です。
SSTOP (T)	スレッドは、明示的な proc(4) 指令またはその他の停止機構により停止しています。
SIDL (I)	スレッドは、プロセス作成中の中間状態です。
SONPROC (O)	CPU 上でスレッドが実行されています。sched:::on-cpu プローブは、スレッドが SONPROC 状態に移行した直後に、SONPROC スレッドのコンテキストで起動します。

psinfo_t

一部の proc プローブは、psinfo_t 型の引数をとります。構造体 psinfo_t については、proc(4) に文書化されています。DTrace コンシューマで使用可能な構造体 psinfo_t の定義は、次のとおりです。

```
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;
```

pr_dmodel フィールドには、32 ビットプロセスであることを示す PR_MODEL_ILP32 と 64 ビットプロセスであることを示す PR_MODEL_LP64 のいずれかが設定されます。

例

exec

exec プローブを使用すると、どのプログラムがだれによって実行されているか、簡単に調べることができます。次の例を参照してください。

```
#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", @);
}
```

ビルドマシン上でしばらくの間このスクリプトを実行すると、次のような出力が得られます。

```
# 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 と exit

プログラムが作成されてから終了するまでの実行時間を調べたい場合は、次の例のように start プローブと exit プローブを有効にします。

```
proc:::start
{
    self->start = timestamp;
}

proc:::exit
```

```
/self->start/
{
    @[execname] = quantize(timestamp - self->start);
    self->start = 0;
}
```

ビルドサーバー上で数秒間このスクリプトを実行すると、次のような出力が得られます。

```
# 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 と lwp-exit

特定のプロセスの実行にかかる時間ではなく、個々のスレッドの実行にかかる時間を調べることもできます。以下の例では、この目的で lwp-start プローブと lwp-exit プローブを使用する方法を紹介します。

```
proc:::lwp-start
/tid != 1/
{
    self->start = timestamp;
}

proc:::lwp-exit
/self->start/
{
    @[execname] = quantize(timestamp - self->start);
    self->start = 0;
}
```

NFS/カレンダサーバー上でこのスクリプトを実行すると、次のような出力が得られます。

```
# 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

シグナルの送受信のプロセスを特定するには、次の例のように signal-send プローブを使用します。

```
#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", @);
}
```

このスクリプトを実行すると、次のような出力が得られます。

```
# 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
```

安定性

以下の表に、proc プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	発展中	発展中	ISA

◆ ◆ ◆ 第 26 章

26 schedプロバイダ

schedプロバイダは、CPUスケジューリング関連のプローブを使用できるようにします。CPUは、すべてのスレッドが使用しなければならない唯一のリソースです。したがって、schedプロバイダは、体系的な動作を理解するためにたいへん役立ちます。たとえば、schedプロバイダを使って、スレッドによるさまざまな処理(スレッドのスリープ、実行、優先順位の変更、別のスレッドの呼び起こしなど)が行われた時期および理由を調べることができます。

プローブ

表 26–1 では、schedプローブについて説明します。

表 26–1 schedプローブ

プローブ	説明
change-pri	スレッドの優先順位が変更される直前に起動するプローブ。args[0]は、スレッドのlwpsinfo_tをポイントしています。スレッドの現在の優先順位は、この構造体のpr_priフィールドに格納されます。args[1]は、このスレッドが含まれているプロセスのpsinfo_tをポイントしています。args[2]には、スレッドの新しい優先順位が格納されます。
dequeue	実行可能スレッドが実行キューから外される直前に起動するプローブ。args[0]は、キューから外されるスレッドのlwpsinfo_tをポイントしています。args[1]は、このスレッドが含まれているプロセスのpsinfo_tをポイントしています。args[2]は、このスレッドがキューから外されるCPUのcpuinfo_tをポイントしています。スレッドが外される実行キューが特定のCPUに関連付けられていない場合、この構造体のcpu_idメンバーの値は-1になります。

表 26–1 sched プローブ (続き)

プローブ	説明
enqueue	実行可能スレッドが実行キューに入れられる直前に起動するプローブ。args[0] は、キューに入れられるスレッドの lwpsinfo_t をポイントしています。args[1] は、このスレッドが含まれているプロセスの psinfo_t をポイントしています。args[2] は、このスレッドがキューに入れられる CPU の cpuinfo_t をポイントしています。スレッドが入れられる実行キューが特定の CPU に関連付けられていない場合、この構造体の cpu_id メンバーの値は -1 になります。args[3] には、スレッドを実行キューの先頭に入れるかどうかを示すブール値が入ります。この値は、スレッドを実行キューの先頭に入れる場合はゼロ以外の値、実行キューの末尾に入れる場合はゼロになります。
off-cpu	現在の CPU がスレッドの実行を終了する直前に起動するプローブ。curcpu 変数は、現在の CPU を表します。curlwpsinfo 変数は、実行を終了しようとしているスレッドを表します。curpsinfo 変数は、現在のスレッドが含まれているプロセスの説明です。args[0] は、現在の CPU が次に実行するスレッドの lwpsinfo_t 構造体をポイントしています。args[1] は、次のスレッドが含まれているプロセスの psinfo_t をポイントしています。
on-cpu	CPU がスレッドの実行を開始した直後に起動するプローブ。curcpu 変数は、現在の CPU を表します。curlwpsinfo 変数は、実行を開始するスレッドを表します。curpsinfo 変数は、現在のスレッドが含まれているプロセスの説明です。
preempt	現在のスレッドが横取りされる直前に起動するプローブ。このプローブが起動すると、現在のスレッドにより、実行するスレッドが選択されます。続いて、現在のスレッドに対して off-cpu プローブが起動します。ある CPU 上のスレッドを横取りするスレッドが、別の CPU 上で実行されている場合もあります。この場合も preempt プローブは起動しますが、ディスパッチャは、実行の優先順位の高いスレッドを見つけることができません。このため、off-cpu プローブではなく remain-cpu プローブが起動します。
remain-cpu	スケジューリングに関する決定がなされたにもかかわらず、現在のスレッドの実行の続行がディスパッチャによって選択されたときに、起動するプローブ。curcpu 変数は、現在の CPU を表します。curlwpsinfo 変数は、実行を開始するスレッドを表します。curpsinfo 変数は、現在のスレッドが含まれているプロセスの説明です。

表 26–1 sched プローブ (続き)

プローブ	説明
schedctl-nopreempt	横取り制御要求により、スレッドが横取りされたあと、実行キューの先頭に入れ直されたときに起動するプローブ。横取り制御の詳細については、schedctl_init(3C) を参照してください。preempt の場合と同じく、schedctl-nopreempt に続いて off-cpu か remain-cpu が起動します。schedctl-nopreempt は、現在のスレッドを実行キューの先頭に入れ直す指示です。このため、off-cpu よりも remain-cpu のほうが、schedctl-nopreempt に続いて起動する確率が高いと言えます。args[0] は、横取りされるスレッドの lwpsinfo_t をポイントしています。args[1] は、このスレッドが含まれているプロセスの psinfo_t をポイントしています。
schedctl-preempt	スレッドが横取り制御を使用しているにもかかわらず横取りされ、実行キューの末尾に入れ直されたときに、起動するプローブ。横取り制御の詳細については、schedctl_init(3C) を参照してください。preempt の場合と同じく、schedctl-preempt に続いて off-cpu か remain-cpu が起動します。preempt の場合と同じく、(そして schedctl-nopreempt の場合とは違って)、schedctl-preempt は、現在のスレッドを実行キューの末尾に入れ直す指示です。その結果、remain-cpu よりも off-cpu のほうが、schedctl-preempt に続いて起動する確率が高いといえます。args[0] は、横取りされるスレッドの lwpsinfo_t をポイントしています。args[1] は、このスレッドが含まれているプロセスの psinfo_t をポイントしています。
schedctl-yield	横取り制御を有効にし、タイムスライスを人工的に延長したスレッドが、CPU をほかのスレッドに譲るコードを実行したときに、起動するプローブ。
sleep	現在のスレッドが同期オブジェクトでスリープする直前に起動するプローブ。同期オブジェクトの型は、curlwpsinfo がポイントしている lwpsinfo_t の pr_stype メンバーに格納されます。同期オブジェクトのアドレスは、curlwpsinfo がポイントしている lwpsinfo_t の pr_wchan メンバーに格納されます。このアドレスの意味は、非公開の実装詳細情報ですが、アドレス値は、同期オブジェクトに固有のトークンとして機能します。
surrender	CPU が、別の CPU から、スケジューリングに関する決定を下すように指示を受けたときに起動するプローブ。こうした指示は、一般に、より優先順位の高いスレッドが実行可能状態になったときに発行されます。
tick	クロック刻みベースのアカウンティングの一環として起動するプローブ。クロック刻みベースのアカウンティングでは、固定間隔の割り込みが発生したとき、どのスレッドとどのプロセスが実行されていたかを調べることによって、CPU のアカウンティングが行われます。args[0] は、CPU 時間の割り当てを受けるスレッドの lwpsinfo_t をポイントしています。args[1] は、このスレッドが含まれているプロセスの psinfo_t をポイントしています。

表 26–1 sched プローブ (続き)

プローブ	説明
wakeup	現在のスレッドが同期オブジェクトでスリープしているスレッドを呼び起こす直前に起動するプローブ。args[0] は、スリープ中のスレッドの lwpsinfo_t をポイントしています。args[1] は、スリープ中のスレッドが含まれているプロセスの psinfo_t をポイントしています。同期オブジェクトの型は、スリープ中のスレッドの lwpsinfo_t の pr_stype メンバーに格納されます。同期オブジェクトのアドレスは、スリープ中のスレッドの lwpsinfo_t の pr_wchan メンバーに格納されます。このアドレスの意味は、非公開の実装詳細情報ですが、アドレス値は、同期オブジェクトに固有のトークンとして機能します。

引数

表 26–2 に、sched プローブの引数の型を一覧します。引数については、表 26–1 を参照してください。

表 26–2 sched プローブ引数

プローブ	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 *	—	—
sleep	—	—	—	—
surrender	lwpsinfo_t *	psinfo_t *	—	—
tick	lwpsinfo_t *	psinfo_t *	—	—
wakeup	lwpsinfo_t *	psinfo_t *	—	—

表 26–2 からわかるように、多くの sched プローブは、引数として lwpsinfo_t のポインタ (スレッド) と psinfo_t のポインタ (スレッドが含まれているプロセス) をとりま

す。これらの構造体の詳細については、それぞれ260 ページの「lwpsinfo_t」と
263 ページの「psinfo_t」を参照してください。

cpuinfo_t

cpuinfo_t 構造体は、CPU を定義します。表 26–2 からわかるように、enqueue プローブと dequeue プローブは、どちらも引数として cpuinfo_t のポインタをとります。また、現在の CPU の cpuinfo_t は、curcpu 変数でポイントされています。以下に、cpuinfo_t 構造体の定義を示します。

```
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;
```

cpu_id メンバーは、psrinfo(1M) と p_online(2) から返されるプロセッサ識別子です。

cpu_pset メンバーは、CPU を含む (CPU がある場合) プロセッサセットです。プロセッサセットの詳細については、psrset(1M) のマニュアルページを参照してください。

cpu_chip メンバーは、物理チップの識別子です。物理チップには、複数の CPU が搭載されていることがあります。詳細については、psrinfo(1M) のマニュアルページを参照してください。

cpu_lgrp メンバーは、CPU の遅延グループの識別子です。遅延グループの詳細については、liblgrp(3LIB) のマニュアルページを参照してください。

cpu_info メンバーは、processor_info(2) から返される CPU の processor_info_t 構造体です。

例

on-cpu と off-cpu

「どの CPU が、どのくらいの時間、スレッドを実行しているか」という問いは頻繁に発生します。この問題へのシステム全体レベルの答えは、次の例のように on-cpu プローブと off-cpu プローブを使用することにより、簡単に導き出すことができます。

```
sched:::on-cpu
{
    self->ts = timestamp;
}

sched:::off-cpu
/self->ts/
{
    @[cpu] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

このスクリプトを実行すると、次のような出力が得られます。

```
# 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
```

出力結果によると、CPU 1 上のスレッドは、100 マイクロ秒未満で一気に実行されるか、あるいは約 10 ミリ秒間実行されます。ヒストグラムから、2 つのデータクラスタ間に顕著な差異があることがわかります。また、特定のプロセスを実行しているCPU を知りたい場合もあります。この場合も、`on-cpu` プローブと `off-cpu` プローブを使用します。以下のスクリプトを実行すると、指定されたアプリケーションを 10 秒間以上実行する CPU が表示されます。

```
#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", @);
}
```

IMAP デーモンを指定して、このスクリプトを大規模メールサーバー上で実行してみましょう。次のような出力が得られます。

```
# 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
```

Solaris は、スレッドを実行する CPU を選択する際、スレッドがスリープしていた時間を考慮します。 スリープ時間が短いスレッドは、移行しにくい傾向にあります。 `off-cpu` プローブと `on-cpu` プローブを使って、この動作を監視できます。

```
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;
}
```

このスクリプトを約 30 秒間実行すると、次の例のような出力が得られます。

```
# 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
```

この出力例では、移行が行われた回数よりも、移行が行われなかった回数のほうが多くなっています。また、スリープ時間が長くなるほど、移行が行われやすくなっています。スリープ時間が 100 ミリ秒以下のときは、分布状態に顕著な差異が見られます。しかし、スリープ時間がもっと長くなれば、この差異はほとんど見られなくなります。この結果から、決められたしきい値を超えると、スケジューリングに関する決定の際にスリープ時間が考慮されなくなるものと推測できます。

最後に、`off-cpu` プローブと `on-cpu` プローブを `pr_stype` フィールドと組み合わせて使用することにより、スレッドがスリープする理由および期間を特定する例を示します。

```
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;
}
```

このスクリプトを数秒間実行すると、次のような出力が得られます。

```
# 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 と dequeue

CPU がアイドル状態になると、ディスパッチャは、別の (アイドル状態でない) CPU のキューにある処理を探します。以下は、dequeue プローブを使って、アプリケーションがどの CPU によって、どのくらいの頻度で転送されるかを確認する例です。

```
#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", @);
}
```

このスクリプトを 4 CPU システム上で実行した場合、出力の末尾部分は次のようになります。

```
# 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
```

どの CPU がどの処理を行なったかではなく、プロセスやスレッドが実行待ち状態になっている CPU について知りたい場合もあります。この場合も、enqueue プローブと dequeue プローブを組み合わせて使用します。

```
sched:::enqueue
{
    self->ts = timestamp;
}

sched:::dequeue
/self->ts/
{
    @[args[2]->cpu_id] = quantize(timestamp - self->ts);
    self->ts = 0;
}
```

このスクリプトを数秒間実行すると、次のような出力が得られます。

```
# 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
```

末尾にゼロ以外の値が出力されている点に注目してください。これらのデータポイントから、どちらの CPU にも、スレッドの実行前に数秒間キューに入れられたインスタンスが、複数あることがわかります。

待ち時間ではなく、実行キューの長さの推移を調べたい場合もあります。このためには、enqueue プローブと dequeue プローブを使って、キューの長さを追跡する連想配列を設定します。

```
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]–;
}
```

ほとんどの場合アイドル状態である単一プロセッサが搭載されたラップトップシステム上で、このスクリプトを約 30 秒間実行すると、次のような出力が得られます。

```
# 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
```

アイドル状態のシステムに典型的な出力結果が得られます。 実行可能スレッドが入れられる時点の実行キューは、ほとんどの場合非常に短いものです (3 スレッド以下)。しかし、このシステムがたいていの場合アイドル状態にあることを考慮すると、表の下部に例外データポイントが出力されるのは、少し予想外かもしれません。たとえば、この実行キューに、実行可能スレッド 13 個分に相当する長さがなぜ必要なのでしょうか。この問題について調べるには、実行キューの長さが長いときに実行キューの内容を表示するような D スクリプトを作成します。この問題を複雑にしているのは、D の有効化です。D の有効化は、データ構造に対して繰り返し実行できないため、当然、実行キュー全体に対しても繰り返し実行できません。たとえ D の有効化にこれが可能であっても、カーネルの内部データ構造への依存関係は避けるべきです。

この種のスクリプトを実行するときは、enqueue プローブと dequeue プローブの両方を有効にし、投機アクションと連想配列の両方を使用します。スレッドがキューに入れられると、スクリプトによって、キューの長さが増分され、スレッドによりキー付けされた連想配列内のタイムスタンプが記録されます。この場合、スレッドは別のスレッドによってキューに入れられた可能性があるので、スレッド固有変数は使用できません。次に、キューの長さが上限を超えていないかどうかのチェックが行われます。上限を超えていた場合、新しい投機アクションが開始され、タイムスタンプと新しい上限値が記録されます。その後、スレッドがキューから外されると、キューに入れられた時点のタイムスタンプと最長のタイムスタンプとの比較が行われます。 スレッドが、最長のタイムスタンプより前にキューに入れられている場合、このスレッドは、最長のタイムスタンプが記録されとき、キュー内にあったことになります。この場合は、このスレッドの情報が投機的にトレースされます。最長のタイムスタンプの時点でキュー内にあったスレッドすべてがカーネルによってキューから外されると、スクリプトにより、投機データのコミットが行われます。以下に、このスクリプトを示します。

```
#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;
}
```

このスクリプトを、先ほどと同じ単一のプロセッサが搭載されたラップトップで実行すると、次のような出力が得られます。

```
# 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
```

出力から、実行キューが長いのは、実行可能な xterm プロセスがたくさんあるせいだということがわかります。この測定と同時に、仮想デスクトップに変更が加えられています。したがって、上のような結果が得られた背景には、何らかの X イベント処理がかかわっていると推測できます。

sleep と wakeup

285 ページの「enqueue と dequeue」の最後の例では、実行可能な xterm プロセスのせいで実行キュー長のバーストが起こったことがわかりました。ここで、この測定結果は仮想デスクトップの変更に起因するものだという仮説を立てることができます。この仮説を検証するには、wakeup プローブを使って、xterm プロセスが何によって呼び起こされているか、またその時期を特定します。次の例を参照してください。

```
#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);
}
```

仮説の検証のため、このスクリプトを実行し、5 秒ほど待ったあと、1 度だけ仮想デスクトップを切り替えます。実行可能な xterm プロセスのバーストが、仮想デスクトップを切り替えたせいで発生しているのであれば、次の出力例のように、5 秒経過したところで呼び起こしアクティビティのバーストが確認されるはずです。

```
# dtrace -s ./xterm.d

  Xsun

           value  ------------- Distribution ------------- count
               4 |                                         0
               5 |@                                        1
               6 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  32
```

```
             7 |                                     0
```

この出力から、xterm プロセスが仮想デスクトップの切り替えを行なった前後の時間帯に集中していることと、これらの xterm プロセスが X サーバーによって呼び起こされたことがわかります。X サーバーと xterm プロセスのやりとりについて調べたい場合は、X サーバーが wakeup プローブを起動したときのユーザースタックトレースを集積します。

X ウィンドウシステムのようなクライアントサーバーシステムのパフォーマンスについて理解するには、サーバーのサービス提供先であるクライアントについて理解する必要があります。この種の問題に対する答えは、従来のパフォーマンス解析ツールでは得にくいものです。しかし、「サーバーにメッセージを送信したクライアントは、サーバー側の処理が完了するまでスリープする」というモデルがある場合は、wakeup プローブを使って、サーバーがどのクライアントの要求を処理しているのかを特定できます。次の例を参照してください。

```
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;
}
```

このスクリプトを実行すると、次のような出力が得られます。

```
dtrace -s ./xwork.d
dtrace: script ’./xwork.d’ matched 14 probes
^C
  xterm                                                     9522510
  soffice.bin                                               9912594
  fvwm2                                                   100423123
  MozillaFirebird                                         312227077
  acroread                                                345901577
```

この出力から、Xsun は、多くの作業を acroread、MozillaFirebird、fvwm2 のために行なっていることがわかります。ただし、fvwm2 の負荷はほかの 2 つほど大きくありません。条件変数同期オブジェクト (SOBJ_CV) からの呼び起こしだけが調べられている点に注目してください。表 25–4 で説明したように、条件変数は同期オブジェクト

の一種であり、通常、共有データ領域へのアクセス以外の理由で同期化を行うときに使用されるものです。X サーバーの場合、クライアントは条件変数でスリープして、パイプ内のデータを待機します。

wakeup プローブとともに、追加で sleep プローブを使用すると、どのアプリケーションがどのアプリケーションでブロックされているのか、どのくらいの間ブロックされているのかを調べることができます。次の例を参照してください。

```
#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", @);
}
```

このスクリプトをデスクトップシステム上で数秒間実行した場合、出力の末尾部分は次のようになります。

```
# 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
```

MozillaFirebird がブロックする様子と、ブロックする理由について調べたいこともあります。この問いの答えを導き出すには、上のスクリプトを以下の例のように変更します。

```
#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", @);
}
```

このスクリプトを数秒間実行すると、次のような出力が得られます。

```
# 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
```

sleep プローブと wakeup プローブを使って、ネームサービスキャッシュデーモンなどの door サーバーのパフォーマンスを調べることもできます。次の例を参照してください。

```
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;
}
```

このスクリプトを大規模メールサーバー上で実行した場合、出力の末尾部分は次のようになります。

```
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
```

automountd の異常なデータポイントや、1 ミリ秒超で持続している sendmail のデータポイントについて調べることもできます。上記のスクリプトに述語を追加して、例外的な (異常な) 結果の原因を絞り込むこともできます。

preempt、remain-cpu

Solaris は横取り式のシステムなので、優先順位の高いスレッドが優先順位の低いスレッドを「横取り」します。横取りが行われると、優先順位の低いほうのスレッド

でかなりの遅延が発生します。このため、どのスレッドがどのスレッドによって横取りされているのか知りたい場合があります。以下は、preempt プローブと remain-cpu プローブを使って、この情報を表示する例です。

```
#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", @);
}
```

このスクリプトをデスクトップシステム上で数秒間実行すると、次のような出力が得られます。

```
# 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

「横取り」は、優先順位に基づいて行われます。この点を考慮して、優先順位が次第に変化していく様子を監視できます。以下は、change-pri プローブを使って、この情報を表示する例です。

```
sched:::change-pri
{
    @[stringof(args[0]->pr_clname)] =
        lquantize(args[2] - args[0]->pr_pri, -50, 50, 5);
}
```

このスクリプトは、優先順位がどの程度上がったか (または下がったか) を確認し、スケジューリングクラス別に結果を集積します。このスクリプトを実行すると、次のような出力が得られます。

```
# 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
```

対話型 (IA) スケジューリングクラスで優先順位の操作が出力されました。優先順位の「操作」ではなく、特定のプロセスとスレッドの優先順位の「値」が変化していく様子を確認することもできます。以下は、`change-pri` プローブを使って、この情報を表示する例です。

```
#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);
}
```

優先順位が変化していく様子を確認するには、ウィンドウに次のコマンドを入力します。

```
$ echo $$
139208
$ while true ; do let i=0 ; done
```

先ほどとは別のウィンドウでスクリプトを実行し、結果をファイルにリダイレクトします。

```
# dtrace -s ./pritime.d 139208 1 > /tmp/pritime.out
#
```

上の例で生成された /tmp/pritime.out ファイルをプロッティングソフトウェアの入力ファイルとして使用すると、優先順位が変化していく様子を視覚的に確認できます。gnuplot は、Solaris Freeware Companion CD に収められている無料版プロッティングパッケージです。gnuplot のデフォルトのインストールディレクトリは、/opt/sfw/bin です。

tick

Solaris では、CPU アカウンティングがクロック刻みで行われます。このような CPU アカウンティングでは、決まった時間間隔でシステムクロック割り込みが発生し、そのとき実行中のスレッドとプロセスが CPU を使用しているものと見なされます。以下は、tick プローブを使ってこのアカウンティングの様子を調べる例です。

```
# 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
```

システムクロック周波数は、オペレーティングシステムによって異なりますが、通常は 25 - 1024 ヘルツです。Solaris のシステムクロック周波数は調整できます。デフォルトでは 100 ヘルツです。

tick プローブは、システムクロックが実行可能なスレッドを検出したときにだけ起動します。tick プローブを使ってシステムクロック周波数を監視するには、常時実行可能なスレッドが必要です。まず、次の例のようなループしたシェルを作成します。

```
$ while true ; do let i=0 ; done
```

先ほどとは別のウィンドウで、次のスクリプトを実行します。

```
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
```

最小の間隔は 9.8 ミリ秒です。つまり、デフォルトのクロック周波数は 10 ミリ秒 (100 ヘルツ) になります。上の例で検出された最小値が 10 ミリ秒未満なのは、ジッターが発生しているせいです。

クロック刻みのアカウンティングには、欠点もあります。それは、アカウンティングを行うシステムクロックによって、時間に関するスケジューリングアクティビティがディスパッチされる場合があるということです。その結果、クロック刻みと同間隔で (つまり 10 ミリ秒に 1 回) スレッドが何らかの作業を行う場合に、正しいアカウンティングができません。時間に関するスケジューリングアクティビティのディスパッチの前後どちらでアカウンティングが行われるかによって、結果が実際よりも上下します。Solaris では、時間に関するディスパッチの前にアカウンティン

グが行われます。その結果、一定間隔で実行されるスレッドについてのアカウンティングが実際より低く報告されます。このようなスレッドは、実行期間をクロック刻みより短くすることによってクロック刻みの背後に「隠す」ことができます。以下は、システムにこのようなスレッドがどのくらいあるかを表示する例です。

```
sched:::tick,
sched:::enqueue
{
    @[probename] = lquantize((timestamp / 1000000) % 10, 0, 10);
}
```

このスクリプトを実行し、出力結果を確認すると、10 ミリ秒間にミリ秒のオフセットが 2 つ存在することがわかります。1 つは tick プローブのオフセット、もう 1 つは enqueue のオフセットです。

```
# 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
```

ヒストグラム tick からは、クロック刻みが 8 ミリ秒のオフセットで起動していることがわかります。スケジューリングがクロック刻みと無関係に行われていれば、enqueue の出力結果は 10 ミリ秒間の中で均一に分布するはずです。しかし、実際には 8 ミリ秒のオフセットで値が大きく変化しています。したがって、このシステムには、時間ベースでスケジューリングされるスレッドも多少存在していることになります。

安定性

以下の表に、sched プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	発展中	発展中	ISA

◆ ◆ ◆ 第 27 章

27 io プロバイダ

io プロバイダは、ディスク入出力に関連したプローブを使用できるようにします。io プロバイダでは、iostat(1M) のような入出力監視ツールで監視している動作をすばやく調べることができます。たとえば、io プロバイダを使用すると、入出力に関する情報を、デバイス別、入出力の種類別、入出力サイズ別、プロセス別、アプリケーション名別、ファイル名別、またはファイルオフセット別に確認できます。

プローブ

表 27–1 は、io プローブの説明です。

表 27–1 io プローブ

プローブ	説明
start	周辺機器 (デバイス) または NFS サーバーに対する入出力要求が発行される直前に起動するプローブ。args[0] は、入出力要求の bufinfo_t をポイントしています。args[1] は、入出力要求の受け取り側デバイスの devinfo_t をポイントしています。args[2] は、入出力要求に対応するファイルの fileinfo_t をポイントしています。ファイル情報を利用できるかどうかは、入出力要求の発行元ファイルシステムによって決まります。詳細については、313 ページの「fileinfo_t」を参照してください。
done	入出力要求が満たされたあと起動するプローブ。args[0] は、入出力要求の bufinfo_t をポイントしています。done プローブは、入出力が完了してから、バッファー上で完了処理が実行されるまでの間に起動します。したがって、done プローブが起動した時点では、b_flags に B_DONE は設定されていません。args[1] は、入出力要求を受け取ったデバイスの devinfo_t をポイントしています。args[2] は、入出力要求に対応するファイルの fileinfo_t をポイントしています。

表 27–1 io プローブ (続き)

プローブ	説明
wait-start	スレッドが入出力要求の完了を待って保留状態に入る直前に起動するプローブ。args[0] は、スレッドが待機する入出力要求の、buf(9S) 構造体をポイントしています。args[1] は、入出力要求を受け取ったデバイスの devinfo_t をポイントしています。args[2] は、入出力要求に対応するファイルの fileinfo_t をポイントしています。wait-start プローブの起動後しばらくすると、同じスレッド内で wait-done プローブが起動します。
wait-done	スレッドが入出力要求の完了待ち状態を抜けたときに起動するプローブ。args[0] は、スレッドが待機する入出力要求の、bufinfo_t をポイントしています。args[1] は、入出力要求を受け取ったデバイスの devinfo_t をポイントしています。args[2] は、入出力要求に対応するファイルの fileinfo_t をポイントしています。wait-done プローブは、同じスレッド内で wait-start プローブが起動したあとでのみ起動します。

io プローブは、周辺機器 に対する入出力要求が発行されたときと、NFS サーバーに対するファイルの読み取り/書き込み要求が発行されたときに起動します。たとえば、NFS サーバーに対してメタデータの要求が発行されても、readdir(3C) によって io プローブがトリガーされることはありません。

引数

表 27–2 に、io プローブの引数の型を示します。引数については、表 27–1 を参照してください。

表 27–2 io プローブ引数

プローブ	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 *

io プローブの引数は、buf(9S) 構造体のポインタ、devinfo_t のポインタ、および fileinfo_t のポインタで構成されます。以下では、これらの構造体について詳しく説明します。

bufinfo_t 構造体

bufinfo_t 構造体は、入出力要求について説明する抽象化オブジェクトです。start プローブ、done プローブ、wait-start プローブ、および wait-done プローブ内の args[0] は、入出力要求のバッファーをポイントしています。以下は、bufinfo_t 構造体の定義です。

```
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;
```

b_flags メンバーは、複数の状態値のビット単位の論理和であり、入出力バッファーの状態を表します。表 27–3 に、有効な状態値を示します。

表 27–3　b_flags の値

B_DONE	データ転送が完了しました。
B_ERROR	入出力転送エラーが発生しました。この値は、b_error フィールドとともに設定されます。
B_PAGEIO	バッファーは、呼び出された入出力要求内で使用されています。詳細については、b_addr フィールドの説明を参照してください。
B_PHYS	バッファーは、ユーザーデータ領域への物理 (直接) 入出力で使用されています。
B_READ	周辺機器から主記憶へデータが読み込まれます。
B_WRITE	主記憶から周辺機器へデータが転送されます。
B_ASYNC	非同期入出力要求です。この要求は待機されません。非同期入出力要求が発行されても、wait-start プローブや wait-done プローブは起動しません。ただし、非同期として指定された一部の入出力要求には、B_ASYNC が設定されていないことがあります。 非同期入出力サブシステムは、別のワークスレッドに非同期入出力操作を行わせることによって、非同期要求に応じることがあります。

b_bcount フィールドには、入出力要求の一環として転送されるバイト数が入ります。

B_PAGEIO が設定されていない場合、b_addr フィールドには、入出力要求の仮想アドレスが入ります。B_PHYS が設定されていない場合、これはカーネル仮想アドレスです。B_PHYS が設定されている場合は、ユーザー仮想アドレスです。B_PAGEIO が設定されている場合、b_addr フィールドには、カーネル非公開データが入ります。B_PHYS と B_PAGEIO は、どちらか 1 つだけを設定してください。そうしないと、両方とも設定されません。

b_lblkno フィールドは、デバイス上のどの論理ブロックがアクセスされるかを表します。論理ブロックから物理ブロック (シリンダ、トラックなど) へのマッピングは、デバイスごとに定義されています。

b_resid フィールドには、エラーのため転送されなかったバイト数が入ります。

b_bufsize フィールドには、割り当て済みのバッファーのサイズが入ります。

b_iodone フィールドは、入出力の完了時に呼び出されるカーネル内の特定のルーチンを表します。

b_error フィールドには、入出力エラーの際ドライバから返されるエラーコードが入ります。b_error が設定されるときは、b_flags メンバー内にも B_ERROR ビットが設定されます。

b_edev フィールドには、アクセス対象のデバイスのメジャーデバイス番号とマイナーデバイス番号が入ります。コンシューマは、D サブルーチン getmajor() と getminor() を使って、b_edev フィールドからメジャーデバイス番号とマイナーデバイス番号を抽出します。

devinfo_t

devinfo_t 構造体は、デバイスに関する情報を提供します。start プローブ、done プローブ、wait-start プローブ、および wait-done プローブ内の args[1] は、入出力のターゲットデバイスの devinfo_t 構造体をポイントしています。devinfo_t は、次のメンバーから成ります。

```
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;
```

dev_major フィールドには、メジャーデバイス番号が入ります。詳細については、getmajor(9F) のマニュアルページを参照してください。

dev_minor フィールドには、マイナーデバイス番号が入ります。詳細については、getminor(9F) のマニュアルページを参照してください。

dev_instance フィールドには、デバイスのインスタンス番号が入ります。デバイスのインスタンス番号は、マイナーデバイス番号とは別のものです。マイナーデバイス番号は、デバイスドライバの管理下にある抽象化オブジェクトです。インスタンス番号は、デバイスノードのプロパティです。デバイスノードのインスタンス番号を表示するには、prtconf(1M) を実行します。

dev_name フィールドには、デバイスを管理するデバイスドライバの名前が入ります。デバイスドライバ名を表示するには、prtconf(1M) に -D を指定して実行します。

dev_statname フィールドには、デバイス名が入ります。このデバイス名は、iostat(1M) を実行したとき出力されるデバイス名と同じです。この名前は、kstat(1M) を実行したとき出力されるカーネル統計情報の名前とも一致しています。このフィールドは、iostat や kstat の出力内容に異常があったとき、この出力を実際の入出力アクティビティとすばやく対応付けるために提供されているものです。

dev_pathname フィールドには、デバイスの完全パスが入ります。このパスを prtconf(1M) の引数に指定すると、詳しいデバイス情報を得ることができます。dev_pathname のパスには、デバイスノード、インスタンス番号、およびマイナーノードを表す要素が含まれています。しかし、統計情報名に、これら 3 つの要素全部が含まれているとはかぎりません。デバイスによっては、統計情報名がデバイス名とインスタンス番号で構成されていることがあります。デバイスによってはまた、デバイス名とマイナーノード番号で構成されていることもあります。その結果、2 つのデバイスの dev_statname は同じでも、dev_pathname は異なるということが起こり得ます。

fileinfo_t

fileinfo_t 構造体は、ファイルに関する情報を提供します。start プローブ、done プローブ、wait-start プローブ、および wait-done プローブの args[2] は、入出力関連のファイルをポイントしています。ファイル情報が存在するかどうかは、入出力要求のディスパッチ時にこの情報を提供するファイルシステムによって決まります。一部のファイルシステム、特に第三者のファイルシステムは、この情報を提供しない場合があります。また、入出力要求の発行元ファイルシステムのファイル情報が存在しない場合もあります。たとえば、ファイルシステムのメタデータへの入出力には、関連ファイルはありません。さらに、高度に最適化されたファイルシステムは、複数の互いに無関係なファイルからの入出力を集積して、単一の入出力要求を作成することがあります。このようなファイルシステムからは、入出力の大部分を表現するファイルまたは入出力の「一部」を表現するファイルのファイル情報が提供されることがあります。また、このようなファイルシステムからファイル情報がまったく提供されないこともあります。

以下に、fileinfo_t 構造体の定義を示します。

```
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;
```

fi_name フィールドには、ファイル名だけが入ります。ここには、ディレクトリ要素は含まれません。入出力に関連ファイル情報がない場合、fi_name フィールドには文字列 <none> が入ります。まれに、ファイルのパス名が未知であることもあります。この場合、fi_name フィールドには文字列 <unknown> が入ります。

fi_dirname フィールドには、ファイル名のディレクトリ要素だけが入ります。fi_name の場合と同じく、ファイル情報が存在しない場合、この文字列は <none> になります。ファイルのパス名が未知である場合、この文字列は <unknown> になります。

fi_pathname フィールドには、ファイルの完全パス名が入ります。fi_name の場合と同じく、ファイル情報が存在しない場合、この文字列は <none> になります。ファイルのパス名が未知である場合、この文字列は <unknown> になります。

fi_offset フィールドには、ファイル内のオフセットが入ります。ファイル情報が存在しない場合や、ファイルシステムがオフセットを指定していない場合は、-1 が入ります。

例

以下は、入出力要求が発行されるたびに関連情報を出力するスクリプトです。

```
#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");
}
```

x86 ラップトップシステム上で Acrobat Reader をコールドスタートした場合、次のような出力が得られます。

```
# 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
       ...
```

入出力が特定のファイル内のデータに関連していない場合、<none> エントリが出力されています。これらの入出力は、任意の書式のメタデータ用です。ファイルのパス名が未知である場合、<unknown> エントリが出力されています。このような事態はめったに発生しません。

連想配列を使って各入出力の所要時間を追跡することにより、上のスクリプト例を少し複雑にすることもできます。次の例を参照してください。

```
#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;
}
```

アイドル状態の x86 ラップトップシステム上で上記スクリプト例を実行し、USB ストレージデバイスを差し込むと、次のような出力が得られます。

```
# 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
  ...
```

この出力から、このシステムの力学についてさまざまな観察が可能です。まず、最初のいくつかの入出力の実行に長い時間がかかっていること (それぞれの所要時間は約 25 ミリ秒) に注目します。これは、`cmdk0` デバイスがラップトップによって電源管理されていたためと考えられます。次に、USB 大容量記憶装置の処理のため scsa2usb(7D) ドライバがロードされた際の入出力を観察します。さらに、この装置が報告される時点で、`/var/adm/messages` への書き込みが発生していることに注目します。最後に、デバイスリンクジェネレータ (ファイル名の末尾が `link.so` のファイル) の読み取りを観察します。デバイスリンクジェネレータは、新しく接続されたデバイスを処理していると考えられます。

`io` プロバイダを使用すると、iostat(1M) 出力について詳しい情報を得ることができます。次の例のような `iostat` 出力を調べるとします。

```
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
```

iotime.d スクリプトを使用して、次の例のように、入出力が行われるたびに確認できます。

```
    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
```

この出力からは、archives.tar というファイルが、cmdk0 によって (/export/archives で) 読み取られており、またデバイス sd2 へ (/mnt で) 書き込まれているかのように見えます。しかし、archives.tar という名前のファイルが 2 つ存在し、並行して別々に処理されるということは通常ありえません。さらに詳しく調べたい場合は、デバイス、アプリケーション、プロセス ID、および転送バイト数を集積します。次の例を参照してください。

```
#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", @);
}
```

このスクリプトを数秒間実行すると、次のような出力が得られます。

```
# dtrace -s ./whoio.d
^C
    DEVICE                  APP        PID           BYTES
     cmdk0                   cp        790         1515520
       sd2                   cp        790         1527808
```

出力から、このアクティビティが、あるデバイスから別のデバイスへの archives.tar ファイルのコピーであったことがわかります。このことから別の疑問が自然にわいてきます:一方のデバイスの速度がもう一方のデバイスよりも高速なのでしょうか。 コピーするときはどちらのデバイスがリミッタになるのでしょうか。こうした問題に答えるには、各デバイスの秒当たりの転送バイト数より、各デバイスの有効なスループットを把握する必要があります。スループットを特定するには、次のようなスクリプトを使用します。

```
#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", @);
}
```

このスクリプトを数秒間実行すると、次のような出力が得られます。

```
  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
```

この出力から、sd2 がリミッタデバイスであることは明白です。sd2 のスループットは毎秒 256 - 512K バイトです。一方、cmdk0 は、毎秒 8M バイトから 64M バイトを超える速さで入出力を行います。このスクリプトは、iostat で出力されるデバイス名と、デバイスの完全パスとの両方を出力します。デバイスについてさらに詳しく知りたい場合は、次の例のように、prtconf にデバイスのパスを指定します。

```
# 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
```

強調部分からわかるように、このデバイスはリムーバブル USB 記憶装置です。

この節の例では、すべての入出力要求について調べてきました。しかし、1 種類の要求だけを調べたい場合もあります。以下の例では、書き込みが発生しているディレクトリと、この書き込みを実行しているアプリケーションを追跡します。

```
#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", @);
}
```

このスクリプトを、デスクトップの作業負荷で一定期間実行すると、次に示すように、興味深い結果が得られます。

```
# 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
```

この出力から、事実上すべての書き込みが Mozilla Firebird のキャッシュと関連付けられていることがわかります。<none> エントリが表示されている書き込みは、おそらく UFS ログに関連付けられている書き込みです。この書き込みは、ファイルシステム上の別の書き込みによって引き起こされます。ロギングの詳細については、ufs(7FS) のマニュアルページを参照してください。この例では、io プロバイダを使って、ソフトウェアの上位層で発生した問題の検出方法を示します。この場合、このスクリプトの構成に問題があります。Web ブラウザのキャッシュが tmpfs(7FS) ファイルシステム内のディレクトリにあれば、この Web ブラウザによる入出力はずっと少なく(おそらく皆無に) なるはずです。

前の例では、start プローブと done プローブしか使われていません。アプリケーションの入出力がブロックされる理由と、その期間を調べたい場合は、wait-start プローブと wait-done プローブを使用します。以下のスクリプトでは、io プローブと sched プローブ (第 26 章「sched プロバイダ」を参照) の両方を使って、StarSuite ソフトウェアの入出力待ち時間と CPU 時間とを比較します。

```
#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);
}
```

StarSuite ソフトウェアのコールドスタート実行中にこのスクリプトを実行すると、次のような結果が得られます。

```
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
```

この出力からわかるように、StarSuite のコールドスタート時間の大部分は入出力待ち時間です。(入出力待ち時間が 13.1 秒であるのに対し、CPU 時間は 3.6 秒。)このスクリプトを StarSuite ソフトウェアのウォームスタートで実行すると、ページキャッシュによって入出力時間が短縮されることがわかります。次の出力例を参照してください。

```
Time breakdown (milliseconds):
  <I/O wait>                                                   0
  <on cpu>                                                  2860

I/O wait breakdown (milliseconds):
```

```
  temp                                                              0
  soffice.tmp                                                       0
  <unknown>                                                         0
  Office                                                            0
```

コールドスタートの出力から、applicat.rdb ファイルが一番入出力待ち時間が長いファイルであることがわかります。これはおそらく、ファイルへの入出力回数が多いためです。このファイルに対して実行された入出力について調べるには、次の D スクリプトを使用します。

```
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);
}
```

このスクリプトは、fileinfo_t 構造体の fi_offset フィールドを使って、ファイルのどの部分がアクセスされているのかをメガバイトの粒度で調べます。StarSuite ソフトウェアのコールドスタート中にこのスクリプトを実行すると、次のような出力が得られます。

```
# 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
```

この出力から、ファイルの最初の 6M バイトだけがアクセスされていることがわかります。これは、おそらく、ファイルサイズが 6M バイトであるためです。また、ファイル全体がアクセスされているわけではないこともわかります。StarSuite のコールドスタートにかかる時間を短縮したい場合は、ファイルのアクセスパターンを把握する必要があります。ファイルの必要な部分がほぼ連続しているなら、StarSuite の実行前にスカウトスレッドを実行して、あらかじめファイルへの入出力を行なっておくと、コールドスタート時間を短縮できます。この方法は、ファイルのアクセスの手段が mmap(2) である場合に特に有効です。しかしながら、この方法で短縮できるコールドスタート時間は約 1.6 秒です。これだけのために、アプリケーションをさらに複雑にし、管理負荷を増やすメリットはありません。どちらにしても、io プロバイダで収集したデータを使って、最終的に得られる利益を正確に推測できます。

安定性

以下の表に、io プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	発展中	発展中	ISA

◆ ◆ ◆ 第 28 章

28 mib プロバイダ

mib プロバイダは、Solaris 管理情報ベース (MIB) 内のカウンタに関連したプローブを提供します。MIB カウンタは、多様なネットワークエンティティをリモートで監視できるようにする簡易ネットワーク管理プロトコル (SNMP) によって使用されます。MIB カウンタは、kstat(1M) コマンドや netstat(1M) コマンドでも表示できます。mib プロバイダを使用すると、リモートまたはローカルのネットワークモニターによって検出されたネットワークの異常な動作について、すばやく調査できます。

プローブ

mib プロバイダは、いくつかの MIB のカウンタ関連のプローブを提供します。表 28–1 に、mib プロバイダによる計測対象の MIB をエクスポートするプロトコルを示します。この表には、一部またはすべての MIB の詳細が記載されている文書の情報、kstat(1M) -n *statistic* コマンドを使って実行カウントにアクセスするときに使用するカーネル統計情報名、およびプローブの完全な定義が記載されている表のタイトルも記載されています。すべての MIB カウンタは、netstat(1M) -s コマンドでも表示できます。

表 28–1 mib プローブ

プロトコル	MIB の説明	カーネル統計情報	mib プローブの表
ICMP	RFC 1213	icmp	表 28–2
IP	RFC 1213	ip	表 28–3
IPsec	—	ip	表 28–4
IPv6	RFC 2465	—	表 28–5
SCTP	「SCTP MIB」(インターネットドラフト)	sctp	表 28–7

表 28–1 mib プローブ (続き)

プロトコル	MIB の説明	カーネル統計情報	mib プローブの表
TCP	RFC 1213	tcp	表 28–8
UDP	RFC 1213	udp	表 28–9

表 28–2 ICMP mib プローブ

icmpInAddrMaskReps	ICMP アドレスマスク応答メッセージが受信されたときに起動するプローブ。
icmpInAddrMasks	ICMP アドレスマスク要求メッセージが受信されたときに起動するプローブ。
icmpInBadRedirects	異常な ICMP リダイレクトメッセージ (例: 未知の ICMP コードが使用されている、送信側または宛先がオフリンクになっているなど) が受信されたときに起動するプローブ。
icmpInCksumErrs	チェックサムが不正な ICMP メッセージが受信されたときに起動するプローブ。
icmpInDestUnreachs	ICMP ターゲットに到達できないというメッセージが受信されたときに起動するプローブ。
icmpInEchoReps	ICMP エコー応答メッセージが受信されたときに起動するプローブ。
icmpInEchos	ICMP エコー要求メッセージが受信されたときに起動するプローブ。
icmpInErrors	ICMP 固有のエラー (例: ICMP チェックサムの不正、長さの不正など) がある ICMP メッセージが受信されたときに起動するプローブ。
icmpInFragNeeded	ICMP ターゲットに到達できない (断片化が必要である) というメッセージが表示されたときに起動するプローブ。送信パケットが MTU よりも長く、かつ Don't Fragment (断片化を行わない) フラグが設定されているために、送信パケットが失われたとき、このメッセージが表示されます。
icmpInMsgs	ICMP メッセージが受信されたときに起動するプローブ。この ICMP メッセージに ICMP 固有のエラーが含まれていると判断された場合、このプローブの起動に続いて、icmpInErrors プローブも起動します。
icmpInOverflows	ICMP メッセージが受信されたあと、バッファーの容量不足のためにそのメッセージが落とされた場合に起動するプローブ。
icmpInParmProbs	ICMP パラメータ問題メッセージが受信されたときに起動するプローブ。
icmpInRedirects	ICMP リダイレクトメッセージが受信されたときに起動するプローブ。
icmpInSrcQuenchs	ICMP 発信元抑制メッセージが受信されたときに起動するプローブ。

表 28–2 ICMP mib プローブ (続き)

icmpInTimeExcds	ICMP 時間切れメッセージが受信されたときに起動するプローブ。
icmpInTimestampReps	ICMP タイムスタンプ応答メッセージが受信されたときに起動するプローブ。
icmpInTimestamps	ICMP タイムスタンプ要求メッセージが受信されたときに起動するプローブ。
icmpInUnknowns	未知の型の ICMP メッセージが受信されたときに起動するプローブ。
icmpOutAddrMaskReps	ICMP アドレスマスク応答メッセージが送信されたときに起動するプローブ。
icmpOutDestUnreachs	ICMP ターゲットに到達できないというメッセージが送信されたときに起動するプローブ。
icmpOutDrops	何らかの理由で発信 ICMP メッセージが落とされたときに起動するプローブ。発信 ICMP メッセージが落とされる理由としては、メモリー割り当てエラーの発生や、不特定多数 (ブロードキャスト) か複数 (マルチキャスト) のソースまたはターゲットの存在などを挙げることができます。
icmpOutEchoReps	ICMP エコー応答メッセージが送信されたときに起動するプローブ。
icmpOutErrors	バッファー不足など、ICMP 層で問題が発見されたため ICMP メッセージが送信されなかったときに起動するプローブ。結果のデータグラムの IP ルーティングを行えないなど、ICMP 層以外でエラーが検出された場合、このプローブは起動しません。
icmpOutFragNeeded	ICMP ターゲットに到達できない (断片化が必要である) というメッセージが送信されたときに起動するプローブ。
icmpOutMsgs	ICMP メッセージが送信されたときに起動するプローブ。この ICMP メッセージに ICMP 固有のエラーが含まれていると判断された場合、このプローブの起動に続いて、icmpOutErrors プローブも起動します。
icmpOutParmProbs	ICMP パラメータ問題メッセージが送信されたときに起動するプローブ。
icmpOutRedirects	ICMP リダイレクトメッセージが送信されたときに起動するプローブ。ホストでは、このプローブは起動しません。これは、ホストがリダイレクトを送信しないからです。
icmpOutTimeExcds	ICMP 時間切れメッセージが送信されたときに起動するプローブ。
icmpOutTimestampReps	ICMP タイムスタンプ応答メッセージが送信されたときに起動するプローブ。

表 28–3 IP mib プローブ

ipForwDatagrams	現在のマシンが、受信したデータグラムの最終的な IP ターゲットでない場合もあります。このプローブは、このような場合に、データグラムを最終的なターゲットへ転送するルートの検索が行われたときに起動します。現在のマシンに IP ゲートウェイ機能がない場合、このプローブは、パケットがこのマシンから始点経路制御されており、かつパケットの始点経路制御オプションの処理に成功したときだけ起動します。
ipForwProhibits	現在のマシンが、受信したデータグラムの最終的な IP ターゲットではなく、ルーターとしても機能しない場合があります。この場合、データグラムを最終的なターゲットへ転送するルートの検索は行われません。このプローブは、このような条件下で起動します。
ipFragCreates	断片化の結果として IP データグラムフラグメントが生成されたときに起動するプローブ。
ipFragFails	断片化できなかった IP データグラムが破棄されたときに起動するプローブ。たとえば、断片化が要求されても Don't Fragment フラグが設定されていると、断片化を行うことができません。
ipFragOKs	IP データグラムが正常に断片化されたときに起動するプローブ。
ipInCksumErrs	IP ヘッダーのチェックサムが不正で、入力データグラムが破棄されたときに起動するプローブ。
ipInDelivers	入力データグラムが IP ユーザープロトコル (ICMP を含む) へ正常に配信されたときに起動するプローブ。
ipInDiscards	パケットとは関係のない理由 (バッファー容量不足など) によって入力 IP データグラムが破棄されたときに起動するプローブ。このプローブは、再アセンブリ待ちのデータグラムが破棄されても起動しません。
ipInHdrErrors	IP ヘッダーのエラーのため入力データグラムが破棄されたときに起動するプローブ。IP ヘッダーのエラーには、バージョン番号の不一致、フォーマットエラー、有効期限切れ、IP オプション処理時のエラーなどがあります。
ipInIPv6	IPv6 パケットが間違って IPv4 キューに到着した場合に起動するプローブ。
ipInReceives	データグラムがインタフェースから受信されたとき (間違って受信された場合も含む) に起動するプローブ。
ipInUnknownProtos	未知のプロトコルまたはサポート対象外のプロトコルを使用したため、ローカルにアドレス指定されたデータグラムが、正常に受信されたあと破棄されたときに起動するプローブ。

表 28–3 IP mib プローブ (続き)

ipOutDiscards	パケットとは関係のない理由 (バッファー容量不足など) によって出力 IP データグラムが破棄されたときに起動するプローブ。このプローブは、パケットが ipForwDatagrams MIB カウンタでカウントされ、かつ任意の破棄条件を満たしている場合に、起動します。
ipOutIPv6	IPv4 接続で IPv6 パケットが送信されたときに起動するプローブ。
ipOutNoRoutes	ターゲットへの転送ルートが見つからず、IP データグラムが破棄されたときに起動するプローブ。このプローブは、パケットが ipForwDatagrams MIB カウンタでカウントされ、かつこの「ルートなし」条件を満たしている場合に、起動します。このプローブは、すべてのデフォルトゲートウェイがダウンしていて、データグラムのルーティングを行えないときにも起動します。
ipOutRequests	ローカルの IP ユーザープロトコル (ICMP を含む) から転送するため、IP に IP データグラムが渡されたときに起動するプローブ。このプローブは、ipForwDatagrams MIB カウンタでパケットがカウントされた場合は起動しません。
ipOutSwitchIPv6	接続で使用する IP プロトコルが IPv4 から IPv6 に変わったときに起動するプローブ。
ipReasmDuplicates	IP 再アセンブリアルゴリズムにより、IP フラグメント内のデータがすべて以前に受信したデータであると判断されたときに起動するプローブ。
ipReasmFails	IP 再アセンブリアルゴリズムによって何らかのエラーが検出されたときに起動するプローブ。このプローブは、IP フラグメントが破棄されるたびに起動するわけではありません。一部のアルゴリズム、特に RFC 815 に記載されているアルゴリズムでは、受信時に複数のフラグメントが組み合わされるので、フラグメントを途中から追跡できなくなることがあるからです。
ipReasmOKs	IP データグラムが正常に再アセンブリされたときに起動するプローブ。
ipReasmPartDups	IP 再アセンブリアルゴリズムにより、IP フラグメント内に以前に受信したデータと新しいデータの両方が含まれていると判断されたときに起動するプローブ。
ipReasmReqds	再アセンブリの必要な IP フラグメントが受信されたときに起動するプローブ。

表 28–4 IPsec mib プローブ

ipsecInFailed	受信したパケットが指定の IPsec ポリシーと一致しないため落とされたときに起動するプローブ。
ipsecInSucceeded	受信したパケットが指定の IPsec ポリシーと一致しているため処理の続行が許可されたときに起動するプローブ。

表 28–5 IPv6 mib プローブ

ipv6ForwProhibits	現在のマシンが、受信した IPv6 データグラムの最終的な IPv6 ターゲットではなく、ルーターとしても機能しない場合があります。この場合、データグラムを最終的なターゲットへ転送するルートの検索は行われません。このプローブは、このような条件下で起動します。
ipv6IfIcmpBadHoplimit	ホップ制限の値が定義済みの最大値未満である ICMPv6 近傍検索プロトコルメッセージが受信されたときに起動するプローブ。こうしたメッセージは、実際に近傍で作成されたものとは限らないため、すべて破棄されます。
ipv6IfIcmpInAdminProhibs	ICMPv6 ターゲットに到達できない (管理者によって通信が禁じられた) というメッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInBadNeighborAdvertisements	異常な ICMPv6 近傍通知メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInBadNeighborSolicitations	異常な ICMPv6 近傍要請メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInBadRedirects	異常な ICMPv6 リダイレクトメッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInDestUnreachs	ICMPv6 ターゲットに到達できないというメッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInEchoReplies	ICMPv6 エコー応答メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInEchos	ICMPv6 エコー要求メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInErrors	ICMPv6 固有のエラー (例: ICMPv6 チェックサムの不正、長さの不正など) がある ICMPv6 メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInGroupMembBadQueries	異常な ICMPv6 グループメンバーシップクエリーメッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInGroupMembBadReports	異常な ICMPv6 グループメンバーシップレポートメッセージが受信されたときに起動するプローブ。

表 28–5 IPv6 mib プローブ (続き)

ipv6IfIcmpInGroupMembOurReports	ICMPv6 グループメンバーシップレポートメッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInGroupMembQueries	ICMPv6 グループメンバーシップクエリーメッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInGroupMembReductions	ICMPv6 グループメンバーシップリダクションメッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInGroupMembResponses	ICMPv6 グループメンバーシップ応答メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInGroupMembTotal	ICMPv6 マルチキャストリスナー探索メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInMsgs	ICMPv6 メッセージが受信されたときに起動するプローブ。このメッセージに ICMPv6 固有のエラーが含まれている場合は、このプローブに続いて、ipv6IfIcmpInErrors プローブも起動します。
ipv6IfIcmpInNeighborAdvertisements	ICMPv6 近傍通知メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInNeighborSolicits	ICMPv6 近傍要請メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInOverflows	ICMPv6 メッセージが受信されたあと、バッファーの容量不足のため落とされたときに起動するプローブ。
ipv6IfIcmpInParmProblems	ICMPv6 パラメータ問題メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInRedirects	ICMPv6 リダイレクトメッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInRouterAdvertisements	ICMPv6 ルーター通知メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInRouterSolicits	ICMPv6 ルーター要請メッセージが受信されたときに起動するプローブ。
ipv6IfIcmpInTimeExcds	ICMPv6 時間切れメッセージが受信されたときに起動するプローブ。
ipv6IfIcmpOutAdminProhibs	ICMPv6 ターゲットに到達できない (管理者によって通信が禁じられた) というメッセージが送信されたときに起動するプローブ。

表 28–5 IPv6 mib プローブ (続き)

ipv6IfIcmpOutDestUnreachs	ICMPv6 ターゲットに到達できないというメッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutEchoReplies	ICMPv6 エコー応答メッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutEchos	ICMPv6 エコーメッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutErrors	バッファー不足など、ICMPv6 層で問題が発見されたため ICMPv6 メッセージが送信されなかったときに起動するプローブ。結果のデータグラムの IPv6 ルーティングを行えないなど、ICMPv6 層以外でエラーが検出された場合、このプローブは起動しません。
ipv6IfIcmpOutGroupMembQueries	ICMPv6 グループメンバーシップクエリーメッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutGroupMembReductions	ICMPv6 グループメンバーシップリダクションメッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutGroupMembResponses	ICMPv6 グループメンバーシップ応答メッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutMsgs	ICMPv6 メッセージが送信されたときに起動するプローブ。このメッセージに ICMPv6 固有のエラーが含まれている場合は、このプローブに続いて、ipv6IfIcmpOutErrors プローブも起動します。
ipv6IfIcmpOutNeighborAdvertisements	ICMPv6 近傍通知メッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutNeighborSolicits	ICMPv6 近傍要請メッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutParmProblems	ICMPv6 パラメータ問題メッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutPktTooBigs	ICMPv6 パケットが大きすぎるというメッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutRedirects	ICMPv6 リダイレクトメッセージが送信されたときに起動するプローブ。ホストでは、このプローブは起動しません。これは、ホストがリダイレクトを送信しないからです。

表 28–5 IPv6 mib プローブ (続き)

ipv6IfIcmpOutRouterAdvertisements	ICMPv6 ルーター通知メッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutRouterSolicits	ICMPv6 ルーター要請メッセージが送信されたときに起動するプローブ。
ipv6IfIcmpOutTimeExcds	ICMPv6 時間切れメッセージが送信されたときに起動するプローブ。
ipv6InAddrErrors	IPv6 ヘッダーの宛先フィールド内の IPv6 アドレスがこのエンティティで受信できる有効なアドレスでないため、入力データグラムが破棄されたときに起動するプローブ。このプローブは、アドレスが不正だった場合 (::0 など) や、サポート対象外だった場合 (接頭辞が割り当てられていないアドレスなど) にも起動します。IPv6 ルーター機能が設定されていないためにデータグラムを転送しないマシンでは、宛先アドレスがローカルアドレスでないとデータグラムが破棄され、このプローブが起動します。
ipv6InDelivers	入力データグラムが IPv6 ユーザープロトコル (ICMPv6 を含む) へ正常に配信されたときに起動するプローブ。
ipv6InDiscards	パケットとは関係のない理由 (バッファー容量不足など) によって入力 IPv6 データグラムが破棄されたときに起動するプローブ。このプローブは、再アセンブリ待ちのデータグラムが破棄されても起動しません。
ipv6InHdrErrors	IPv6 ヘッダーのエラーのため入力データグラムが破棄されたときに起動するプローブ。IPv6 ヘッダーのエラーには、バージョン番号の不一致、フォーマットエラー、ホップカウントの超過、IPv6 オプション処理時のエラーなどがあります。
ipv6InIPv4	IPv4 パケットが間違って IPv6 キューに到着した場合に起動するプローブ。
ipv6InMcastPkts	マルチキャスト IPv6 パケットが受信されたときに起動するプローブ。
ipv6InNoRoutes	ターゲットへの転送ルートが見つからず、経路指定された IPv6 データグラムが破棄されたときに起動するプローブ。このプローブが起動するのは、パケットが外部から発信された場合だけです。

表 28–5 IPv6 mib プローブ (続き)

ipv6InReceives	IPv6 データグラムがインタフェースから受信されたとき (間違って受信された場合も含む) に起動するプローブ。
ipv6InTooBigErrors	最大フラグメントサイズを超えるサイズのフラグメントが受信されたときに起動するプローブ。
ipv6InTruncatedPkts	入力データグラムフレームに十分なデータが含まれていないため破棄されたときに起動するプローブ。
ipv6InUnknownProtos	未知のプロトコルまたはサポート対象外のプロトコルを使用したため、ローカルにアドレス指定された IPv6 データグラムが、正常に受信されたあと破棄されたときに起動するプローブ。
ipv6OutDiscards	パケットとは関係のない理由 (バッファー容量不足など) によって出力 IPv6 データグラムが破棄されたときに起動するプローブ。このプローブは、パケットが ipv6OutForwDatagrams MIB カウンタでカウントされ、かつ任意の破棄条件を満たしている場合に、起動します。
ipv6OutForwDatagrams	現在のマシンが、受信したデータグラムの最終的な IPv6 ターゲットでない場合もあります。このプローブは、データグラムを最終的なターゲットへ転送するルートの検索が行われたときに起動します。現在のマシンに IPv6 ルーター機能がない場合、このプローブは、パケットがこのマシンから始点経路制御されており、かつパケットの始点経路制御オプションの処理に成功したときだけ起動します。
ipv6OutFragCreates	断片化の結果として IPv6 データグラムフラグメントが生成されたときに起動するプローブ。
ipv6OutFragFails	断片化できなかった IPv6 データグラムが破棄されたときに起動するプローブ。たとえば、Don't Fragment フラグが設定されていると、断片化を行うことができません。
ipv6OutFragOKs	IPv6 データグラムが正常に断片化されたときに起動するプローブ。
ipv6OutIPv4	IPv4 接続で IPv6 パケットが送信されたときに起動するプローブ。
ipv6OutMcastPkts	マルチキャストパケットが送信されたときに起動するプローブ。

表 28–5 IPv6 mib プローブ (続き)

ipv6OutNoRoutes	ターゲットへの転送ルートが見つからず、IPv6 データグラムが破棄されたときに起動するプローブ。このプローブは、パケットが外部から発信された場合は起動しません。
ipv6OutRequests	ローカルの IPv6 ユーザープロトコル (ICMPv6 を含む) から転送するため、IPv6 に IPv6 データグラムが渡されたときに起動するプローブ。このプローブは、ipv6ForwDatagrams MIB カウンタでパケットがカウントされた場合は起動しません。
ipv6OutSwitchIPv4	接続で使用する IP プロトコルが IPv6 から IPv4 に変わったときに起動するプローブ。
ipv6ReasmDuplicates	IPv6 再アセンブリアルゴリズムにより、IPv6 フラグメント内のデータがすべて以前に受信したデータであると判断されたときに起動するプローブ。
ipv6ReasmFails	IPv6 再アセンブリアルゴリズムによって何らかのエラーが検出されたときに起動するプローブ。このプローブは、IPv6 フラグメントが破棄されるたびに起動するわけではありません。一部のアルゴリズムでは、受信時に複数のフラグメントが組み合わされるので、フラグメントを途中から追跡できなくなることがあるからです。
ipv6ReasmOKs	IPv6 データグラムが正常に再アセンブリされたときに起動するプローブ。
ipv6ReasmPartDups	IPv6 再アセンブリアルゴリズムにより、IPv6 フラグメント内に以前に受信したデータと新しいデータの両方が含まれていると判断されたときに起動するプローブ。
ipv6ReasmReqds	再アセンブリの必要な IPv6 フラグメントが受信されたときに起動するプローブ。

表 28–6 生の IP mib プローブ

rawipInCksumErrs	IP チェックサムが不正な、生の IP パケットが受信されたときに起動するプローブ。
rawipInDatagrams	生の IP パケットが受信されたときに起動するプローブ。
rawipInErrors	異常な、生の IP パケットが受信されたときに起動するプローブ。
rawipInOverflows	生の IP パケットが受信されたあと、バッファーの容量不足のため落とされたときに起動するプローブ。
rawipOutDatagrams	生の IP パケットが送信されたときに起動するプローブ。

表 28–6 生の IP mib プローブ (続き)

rawipOutErrors	何らかのエラー条件の発生によって生の IP パケットが送信されなかったときに起動するプローブ。たとえば、生の IP パケットに異常がある場合、このパケットは送信されません。

表 28–7 SCTP mib プローブ

sctpAborted	SCTP 結合が、結合の異常な終了を意味する ABORT プリミティブを使って、ある状態から CLOSED 状態へ直接移行したときに起動するプローブ。
sctpActiveEstab	SCTP 結合が、COOKIE-ECHOED 状態から、結合処理を最初に試みたのが上位層であることを意味する ESTABLISHED 状態へ直接移行したときに起動するプローブ。
sctpChecksumError	チェックサムの無効な SCTP パケットがピアから受信されたときに起動するプローブ。
sctpCurrEstab	sctpCurrEstab MIB カウンタの読み取りの一環として SCTP 結合にしるしが付けられたときに起動するプローブ。SCTP 結合にしるしが付けられるのは、現在の状態が ESTABLISHED、SHUTDOWN-RECEIVED、SHUTDOWN-PENDING のいずれかである場合です。
sctpFragUsrMsgs	MTU のためユーザーメッセージを断片化する必要があるときに起動するプローブ。
sctpInClosed	終了した SCTP 結合でデータが受信されたときに起動するプローブ。
sctpInCtrlChunks	sctpInCtrlChunks MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。
sctpInDupAck	重複した ACK が受信されたときに起動するプローブ。
sctpInInvalidCookie	無効な Cookie が受信されたときに起動するプローブ。
sctpInOrderChunks	sctpInOrderChunks MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。
sctpInSCTPPkts	sctpInSCTPPkts MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。
sctpInUnorderChunks	sctpInUnorderChunks MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。

表 28–7 SCTP mib プローブ (続き)

sctpListenDrop	何らかの理由で着信接続が落とされたときに起動するプローブ。
sctpOutAck	選択的肯定応答が送信されたときに起動するプローブ。
sctpOutAckDelayed	SCTP 結合の遅延肯定応答が処理されたときに起動するプローブ。遅延肯定応答処理の一環として送信される肯定応答は、sctpOutAck プローブを起動します。
sctpOutCtrlChunks	sctpOutCtrlChunks MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。
sctpOutOfBlue	どの結合に属するかを受信側で識別できないが、これ以外には問題のない SCTP パケットが受信されたときに起動するプローブ。
sctpOutOrderChunks	sctpOutOrderChunks MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。
sctpOutSCTPPkts	sctpOutSCTPPkts MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。
sctpOutUnorderChunks	sctpOutUnorderChunks MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。
sctpOutWinProbe	ウィンドウプローブが送信されたときに起動するプローブ。
sctpOutWinUpdate	ウィンドウ更新が送信されたときに起動するプローブ。
sctpPassiveEstab	SCTP 結合が CLOSED 状態から ESTABLISHED 状態へ直接移行したときに起動するプローブ。この結合の試みは、リモート端点によって開始されました。
sctpReasmUsrMsgs	sctpReasmUsrMsgs MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。
sctpRetransChunks	sctpRetransChunks MIB カウンタが更新されたときに起動するプローブ。このカウンタは、明示的に照会された場合、または SCTP 接続が終了した場合に更新されます。args[0] には、MIB カウンタの値の増分値が入ります。

表 28–7 SCTP mib プローブ (続き)

sctpShutdowns	SCTP 結合が、SHUTDOWN-SENT 状態または SHUTDOWN-ACK-SENT 状態から CLOSED 状態へ直接移行したときに起動するプローブ。この移行は、結合が正常に終了したことを表します。
sctpTimHeartBeatDrop	ハートビート通知が受信できなかったため SCTP 結合が異常終了したときに起動するプローブ。
sctpTimHeartBeatProbe	SCTP ハートビートが送信されたときに起動するプローブ。
sctpTimRetrans	結合に対してタイマーベースの再転送処理が実行されたときに起動するプローブ。
sctpTimRetransDrop	タイマーベースの再転送の実行に長い間成功しない状態が続いたため、結合が異常終了したときに起動するプローブ。

表 28–8 TCP mib プローブ

tcpActiveOpens	TCP 接続が CLOSED 状態から SYN_SENT 状態へ直接移行したときに起動するプローブ。
tcpAttemptFails	TCP 接続が SYN_SENT 状態または SYN_RCVD 状態から CLOSED 状態へ直接移行したときと、TCP 接続が SYN_RCVD 状態から LISTEN 状態へ直接移行したときに起動するプローブ。
tcpCurrEstab	tcpCurrEstab MIB カウンタの読み取りの一環として TCP 接続にしるしが付けられたときに起動するプローブ。TCP 接続にしるしが付けられるのは、現在の状態が ESTABLISHED または CLOSE_WAIT である場合です。
tcpEstabResets	TCP 接続が ESTABLISHED 状態または CLOSE_WAIT 状態から CLOSED状態へ直接移行したときに起動するプローブ。
tcpHalfOpenDrop	SYN_RCVD 状態にある接続キューがいっぱいになったため接続が落とされたときに起動するプローブ。
tcpInAckBytes	以前に送信されたデータの ACK が受信されたときに起動するプローブ。args[0] には、通知のバイト数が入ります。
tcpInAckSegs	以前に送信されたセグメントの ACK が受信されたときに起動するプローブ。
tcpInAckUnsent	送信されていないセグメントの ACK が受信されたときに起動するプローブ。
tcpInClosed	終了状態の接続がデータを受信したときに起動するプローブ。
tcpInDataDupBytes	受信されたセグメントに、以前に受信済みのデータだけが含まれているとき、起動するプローブ。args[0] には、重複したセグメント内のバイト数が入ります。

表 28–8 TCP mib プローブ (続き)

tcpInDataDupSegs	受信されたセグメントに、以前に受信済みのデータだけが含まれているとき、起動するプローブ。args[0] には、重複したセグメント内のバイト数が入ります。
tcpInDataInorderBytes	新しいデータのシーケンス番号より前のすべてのデータが以前に受信されているときに起動するプローブ。args[0] には、順番に受信されたバイト数が入ります。
tcpInDataInorderSegs	新しいセグメントのシーケンス番号より前のすべてのセグメントが以前に受信されているときに起動するプローブ。
tcpInDataPartDupBytes	受信されたセグメントに、以前に受信済みのデータと新しいデータの両方が含まれているとき、起動するプローブ。args[0] には、重複バイト数が入ります。
tcpInDataPartDupSegs	受信されたセグメントに、以前に受信済みのデータと新しいデータの両方が含まれているとき、起動するプローブ。args[0] には、重複バイト数が入ります。
tcpInDataPastWinBytes	現在の受信ウィンドウを超えるデータが受信されたときに起動するプローブ。args[0] には、バイト数が入ります。
tcpInDataPastWinSegs	現在の受信ウィンドウを超えるセグメントが受信されたときに起動するプローブ。
tcpInDataUnorderBytes	新しいデータのシーケンス番号より前のデータの一部が不足しているときに起動するプローブ。args[0] には、ランダムに受信されたバイト数が入ります。
tcpInDataUnorderSegs	新しいデータのシーケンス番号より前のデータの一部が不足しているときに起動するプローブ。
tcpInDupAck	重複した ACK が受信されたときに起動するプローブ。
tcpInErrs	受信セグメントに TCP エラー (不正な TCP チェックサムなど) が見つかったときに起動するプローブ。
tcpInSegs	セグメントが受信されたときに起動するプローブ。あとで、このセグメントにエラーが含まれていて、以降の処理を続行できないことがわかる場合もあります。
tcpInWinProbe	ウィンドウプローブが受信されたときに起動するプローブ。
tcpInWinUpdate	ウィンドウ更新が受信されたときに起動するプローブ。
tcpListenDrop	待機キューがいっぱいになったため着信接続が落とされたときに起動するプローブ。
tcpListenDropQ0	SYN_RCVD 状態にある接続キューがいっぱいになったため接続が落とされたときに起動するプローブ。
tcpOutAck	ACK が送信されたときに起動するプローブ。

表 28–8 TCP mib プローブ (続き)

tcpOutAckDelayed	最初の遅延のあと ACK が送信されたときに起動するプローブ。
tcpOutControl	SYN、FIN、RST のいずれかが送信されたときに起動するプローブ。
tcpOutDataBytes	データが送信されたときに起動するプローブ。args[0] には、送信バイト数が入ります。
tcpOutDataSegs	セグメントが送信されたときに起動するプローブ。
tcpOutFastRetrans	高速再転送アルゴリズムの一環としてセグメントが再転送されたときに起動するプローブ。
tcpOutRsts	RST フラグが設定されているセグメントが送信されたときに起動するプローブ。
tcpOutSackRetransSegs	選択的肯定応答が有効にされている接続でセグメントが再転送されたときに起動するプローブ。
tcpOutSegs	再転送されたのではないバイトが 1 バイト以上含まれているセグメントが送信されたときに起動するプローブ。
tcpOutUrg	URG フラグが設定されていて、有効な緊急ポインタを持つセグメントが送信されたときに起動するプローブ。
tcpOutWinProbe	ウィンドウプローブが送信されたときに起動するプローブ。
tcpOutWinUpdate	ウィンドウ更新が送信されたときに起動するプローブ。
tcpPassiveOpens	TCP 接続が LISTEN 状態から SYN_RCVD 状態へ直接移行したときに起動するプローブ。
tcpRetransBytes	データが再転送されたときに起動するプローブ。args[0] には、再転送されたバイト数が入ります。
tcpRetransSegs	再転送されたバイトが 1 バイト以上含まれているセグメントが送信されたときに起動するプローブ。
tcpRttNoUpdate	データが受信されたが、RTT を更新するためのタイムスタンプ情報がないときに起動するプローブ。
tcpRttUpdate	RTT の更新に必要なタイムスタンプ情報を含むデータが受信されたときに起動するプローブ。
tcpTimKeepalive	接続に対してタイマーベースのキープアライブ処理が実行されたときに起動するプローブ。
tcpTimKeepaliveDrop	キープアライブ処理の結果、接続が終了したときに起動するプローブ。
tcpTimKeepaliveProbe	キープアライブ処理の一環としてキープアライブプローブが送信されたときに起動するプローブ。
tcpTimRetrans	接続に対してタイマーベースの再転送処理が実行されたときに起動するプローブ。

表 28–8 TCP mib プローブ (続き)

tcpTimRetransDrop	タイマーベースの再転送の実行に長い間成功しない状態が続いたため、接続が終了したときに起動するプローブ。

表 28–9 UDP mib プローブ

udpInCksumErrs	UDP チェックサムが不正で、データグラムが破棄されたときに起動するプローブ。
udpInDatagrams	UDP データグラムが受信されたときに起動するプローブ。
udpInErrors	受信された UDP データグラムが、パケットヘッダーの異常や内部バッファーの割り当てエラーのため破棄されたときに起動するプローブ。
udpInOverflows	UDP データグラムが受信されたあと、バッファーの容量不足のため落とされたときに起動するプローブ。
udpNoPorts	ソケットが結合されていないポートで UDP データグラムが受信されたときに起動するプローブ。
udpOutDatagrams	UDP データグラムが送信されたときに起動するプローブ。
udpOutErrors	何らかのエラー条件の発生によって UDP データグラムが送信されなかったときに起動するプローブ。たとえば、異常な UDP データグラムは送信されません。

引数

mib プローブ用の唯一の引数は、すべてのプローブに共通の意味を持っています。args[0] には、カウンタの増分値が入ります。ほとんどの mib プローブでは、args[0] の値は常に 1 ですが、任意の正の値が入る場合もあります。このようなプローブの args[0] の意味については、プローブの説明に記載されています。

安定性

以下の表に、mib プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明

要素	名前の安定性	データの安定性	依存クラス
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	発展中	発展中	ISA

◆ ◆ ◆ 第 29 章

29 fpuinfo プロバイダ

fpuinfo プロバイダは、SPARC マイクロプロセッサで行われる浮動小数点命令のシミュレーションに関連するプローブを提供します。ほとんどの浮動小数点命令はハードウェアで実行されますが、一部の浮動小数点命令はオペレーティングシステムをトラップし、ここでシミュレーションが行われます。どのような条件のときに、浮動小数点演算のシミュレーションがオペレーティングシステム側で行われるかは、マイクロプロセッサの実装ごとに異なります。シミュレーションを必要とする演算はごく少数です。しかし、このような演算を頻繁に行うアプリケーションがある場合、パフォーマンスに深刻な影響が現れる可能性があります。fpuinfo プロバイダでは、kstat(1M) と fpu_info カーネル統計情報、または trapstat(1M) と fp-xcp-other トラップを利用して、浮動小数点演算のシミュレーション内容をすばやく確認できます。

プローブ

fpuinfo プロバイダは、シミュレート可能な浮動小数点命令の型ごとに 1 つずつ、プローブを提供します。fpuinfo プロバイダでは、CPU の「名前の安定性」が実現されています。したがって、プローブ名はマイクロプロセッサの実装ごとに固有であり、同じファミリに属する別のマイクロプロセッサ上では使用できないことがあります。たとえば、以下の表に示すプローブの中には、UltraSPARC-III では使用できるが UltraSPARC-III+ では使用できないものもあれば、その逆もあります。

Table 29–1 に、表 29–1 プローブを一覧します。

表 29–1 fpuinfo プローブ

fpu_sim_fitoq	カーネルが fitoq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fitod	カーネルが fitod 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fitos	カーネルが fitos 命令をシミュレートしたときに起動するプローブ。

表 29–1 fpuinfo プローブ (続き)

fpu_sim_fxtoq	カーネルが fxtoq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fxtod	カーネルが fxtod 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fxtos	カーネルが fxtos 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fqtox	カーネルが fqtox 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fdtox	カーネルが fdtox 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fstox	カーネルが fstox 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fqtoi	カーネルが fqtoi 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fdtoi	カーネルが fdtoi 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fstoi	カーネルが fstoi 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fsqrtq	カーネルが fsqrtq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fsqrtd	カーネルが fsqrtd 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fsqrts	カーネルが fsqrts 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fcmpeq	カーネルが fcmpeq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fcmped	カーネルが fcmped 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fcmpes	カーネルが fcmpes 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fcmpq	カーネルが fcmpq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fcmpd	カーネルが fcmpd 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fcmps	カーネルが fcmps 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fdivq	カーネルが fdivq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fdivd	カーネルが fdivd 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fdivs	カーネルが fdivs 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fdmulx	カーネルが fdmulx 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fsmuld	カーネルが fsmuld 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fmulq	カーネルが fmulq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fmuld	カーネルが fmuld 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fmuls	カーネルが fmuls 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fsubq	カーネルが fsubq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fsubd	カーネルが fsubd 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fsubs	カーネルが fsubs 命令をシミュレートしたときに起動するプローブ。

表 29–1 fpuinfo プローブ (続き)

fpu_sim_faddq	カーネルが faddq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_faddd	カーネルが faddd 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fadds	カーネルが fadds 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fnegd	カーネルが fnegd 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fnegq	カーネルが fneqq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fnegs	カーネルが fnegs 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fabsd	カーネルが fabsd 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fabsq	カーネルが fabsq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fabss	カーネルが fabss 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fmovd	カーネルが fmovd 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fmovq	カーネルが fmovq 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fmovs	カーネルが fmovs 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fmovr	カーネルが fmovr 命令をシミュレートしたときに起動するプローブ。
fpu_sim_fmovcc	カーネルが fmovcc 命令をシミュレートしたときに起動するプローブ。

引数

fpuinfo プローブは引数をとりません。

安定性

以下の表に、fpuinfo プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	CPU
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	CPU
引数	発展中	発展中	CPU

◆ ◆ ◆ 第 30 章

pid プロバイダ

pid プロバイダでは、ユーザープロセス内の関数の開始 (entry) と終了 (return) のトレースと、絶対アドレスまたは関数オフセットで指定した命令のトレースを行うことができます。プローブが有効にされていなければ、pid プロバイダの及ぼすプローブの影響は皆無です。プローブが有効にされていれば、トレース対象のプロセスだけがプローブの影響を受けます。

注 – コンパイラによって関数がインライン展開されても、pid プロバイダのプローブは起動されません。コンパイル時に関数がインライン展開されないようにするには、コンパイラのマニュアルを参照してください。

注 – 関数ポインタを使ってサブ関数を呼び出す関数を pid プロバイダがプローブすると、予期しない動作になります。関数に入るときのアドレスと関数から戻るときのアドレスに明示的にプローブを置くことで、そのような関数を解析することができます。

---

pid プローブの命名

pid プロバイダは、実際は、複数のプロバイダから成る「クラス」を定義します。プロセスごとに固有の pid プロバイダを関連付けることができます。たとえばプロセス ID 123 のプロセスをトレースしたい場合は、pid123 プロバイダを使用します。このようなプロバイダから提供されるプローブでは、プローブ記述のモジュール部分が、対応するプロセスのアドレス空間にロードされたオブジェクトを示します。以下は、mdb(1) を使ってオブジェクトのリストを表示する例です。

```
$ 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
```

プローブ記述内では、ファイル名 (完全パス名ではない) でオブジェクトに名前を付けます。接尾辞 .1、so.1 は省略することもできます。以下の例では、すべて同じプローブに名前を付けています。

```
pid123:libc.so.1:strcpy:entry
pid123:libc.so:strcpy:entry
pid123:libc:strcpy:entry
```

最初の例はプローブの実際の名前です。残りの 2 つの例は別名です。これらの別名は、内部で完全ロードオブジェクト名に置き換えられます。

実行可能ファイルのロードオブジェクトには、別名 a.out を使用できます。次の 2 つのプローブ記述は、同じプローブに名前を付けています。

```
pid123:csh:main:return
pid123:a.out:main:return
```

アンカーされた DTrace プローブの場合と同様に、プローブ記述の関数フィールドを使って、モジュールフィールド内の関数に名前を付けます。ユーザーアプリケーションバイナリでは、同じ関数に複数の名前が付けられていることがあります。たとえば、mutex_lock は libc.so.1 内の関数 pthread_mutex_lock の代替名になります。DTrace は、このような関数に対して正規名を 1 つ選択し、これを内部で使用します。以下の例では、DTrace がモジュール名と関数名を正規の形式に再マッピングする内部手続きの様子を確認できます。

```
# 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
```

この自動リネームにより、有効にする前と実際に有効にしたあとで、プローブ名が若干変化することがあります。同じ Solaris リリースを実行しているシステム同士であれば、DTrace をいつ実行しても、正規名は常に同じです。

pid プロバイダの効果的な使用例については、第 33 章「ユーザープロセスのトレース」を参照してください。

関数境界プローブ

FBT プロバイダを使用するとカーネル内の関数の開始 (entry) と終了 (return) をトレースすることができるのと同様に、pid プロバイダには、ユーザープログラム内の関数の開始 (entry) と終了 (return) をトレースする機能があります。このマニュアルでは、FBT プロバイダを使ってカーネル関数呼び出しをトレースする例をいくつか紹介していますが、これらの例をほんの少し変更するだけでユーザープロセスにも適用することができます。

entry プローブ

entry プローブは、トレース対象の関数が呼び出されたときに起動します。entry プローブの引数は、トレース対象の関数の引数の値と一致します。

return プローブ

return プローブは、トレース対象の関数が復帰したとき、または別の関数の末尾呼び出しを行なったときに起動します。arg0 には復帰命令の関数のオフセット、arg1 には戻り値が入ります。

注 – arg*N* を使用すると、フィルタリングされていない生の値が型 int64_t として返されます。pid プロバイダでは args[*N*] 書式はサポートされません。

関数オフセットプローブ

pid プロバイダを使用すると、関数内のあらゆる命令をトレースできます。たとえば、関数 main() 内に 4 バイトの命令をトレースするには、次のようなコマンドを使用します。

```
pid123:a.out:main:4
```

このプローブは、main+4 のアドレスでこの命令が実行されるたびに起動します。オフセットプローブの引数は未定義です。これらのプローブが設定されている位置でプロセスの状態を確認したい場合は、uregs[] 配列が便利です。詳細については、366 ページの「uregs[] 配列」を参照してください。

安定性

以下の表に、pid プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	非公開	非公開	不明 (Unknown)

◆ ◆ ◆ 第 31 章

plockstat プロバイダ

plockstat プロバイダが提供するプローブを使用すると、ユーザーレベルの同期プリミティブの動作 (ロック競合やロック保持時間など) を観察できます。plockstat(1M) コマンドは DTrace コンシューマの一種であり、plockstat プロバイダを利用して、ユーザーレベルのロックイベントに関するデータを収集します。

概要

plockstat プロバイダは、次のようなイベントに対応したプローブを提供します。

競合イベント　これらのプローブは、ユーザーレベルの同期プリミティブの競合に関するプローブです。リソースが使えるようになるまで、スレッドが強制的に待機させられるときに起動します。通常、Solaris は、競合が起こっていない状態に合わせて最適化されています。長期にわたる競合状態は想定されていません。これらのプローブは、実際に競合が発生したとき、その状況を把握するために使用してください。競合は比較的まれにしか起こりません。このため、通常は、競合イベントプローブを有効にしたからといって、プローブの深刻な影響はありません。これらのプローブを有効にするとき、パフォーマンスへの影響を心配する必要はありません。

保持イベント　これらのプローブは、ユーザーレベルの同期プリミティブの獲得や解放などの操作に関連しています。これらのプローブは、ユーザーレベルの同期プリミティブの操作方法に関する問題に答えるために使用します。一般に、アプリケーションは、同期プリミティブの獲得と解放を非常に頻繁に行います。したがって、保持イベントプローブを有効にしたときのほうが、競合イベントプローブを有効にしたときよりも、プローブの及ぼす影響は大きくなります。プローブを有効にしたときの影響は甚大になりえますが、決して異常なものではありません。本稼働アプリケーションでプローブを有効にしても、まったく問題はありません。

エラーイベント　これらのプローブは、ユーザーレベルの同期プリミティブの獲得または解放時に起こる異常な動作に関連したプローブです。これらのイベントを利用して、ユーザーレベルの同期プリミティブでスレッドがブロックしているときに発生したエラーを検出できます。エラーイベントは非常にまれにしか発生しないので、エラーイベントプローブを有効にしても、プローブの深刻な影響はありません。

相互排他ロックプローブ

「相互排他ロック」は、クリティカルセクションの相互排他を行います。あるスレッドから、`mutex_lock(3C)` や `pthread_mutex_lock(3C)` を使って、別のスレッドが保持している相互排他ロックを獲得できます。このとき、ロックを獲得しようとしているスレッドは、ロックを所有しているスレッドが別の CPU 上で実行されているかどうかを確認します。ロックの所有スレッドが別の CPU 上で実行されている場合、ロックを獲得しようとしているスレッドは、相互排他ロックが使用可能な状態になるまでしばらく「スピン」して待機します。所有スレッドが別の CPU 上で実行されていない場合、ロックを獲得しようとしているスレッドは「ブロック」されます。

表 31–1 に、相互排他ロック関連の 4 種類の `plockstat` プローブを示します。それぞれの `arg0` には、相互排他ロックを表す `mutex_t` または `pthread_mutex_t` 構造体 (型はすべて同じ) のポインタが入ります。

表 31–1　相互排他ロックプローブ

`mutex-acquire`	相互排他ロックの獲得後すぐに起動する保持イベントプローブ。`arg1` には、獲得したロックが再帰型相互排他ロックで再帰的に機能するかどうかを示すブール値が入ります。`arg2` には、ロックを獲得しようとしているスレッドがこの相互排他ロック上でスピンする回数が入ります。この相互排他ロックの獲得時に `mutex-spin` プローブが起動している場合にのみ、`arg2` にはゼロ以外の値が入ります。
`mutex-block`	保持されている相互排他ロックでスレッドがブロックされる前に起動する競合イベントプローブ。1 回のロック獲得で `mutex-block` と `mutex-spin` の両者が起動することもあります。
`mutex-spin`	保持されている相互排他ロックでスレッドがスピンを開始する前に起動する競合イベントプローブ。1 回のロック獲得で `mutex-block` と `mutex-spin` の両者が起動することもあります。
`mutex-release`	相互排他ロックの解放後すぐに起動する保持イベントプローブ。`arg1` には、このイベントが再帰型相互排他ロックでの再帰的な解放に当たるかどうかを示すブール値が入ります。
`mutex-error`	相互排他処理でエラーが発生したときに起動するエラーイベントプローブ。`arg1` には、発生したエラーの `errno` 値が入ります。

読み取り/書き込みロックプローブ

「読み取り/書き込みロック」では、同一クリティカルセクション内で同時に複数の読み取り、または 1 回の書き込みを行うことができます。両方を同時に行うことはできません。通常、これらのロックを使用するのは、変更されるより検索される機会のほうが多い構造体や、クリティカルセクションに長時間を費やすスレッドです。読み取り/書き込みロックの操作は、Solaris の `rwlock(3C)` または POSIX の `pthread_rwlock_init(3C)` インタフェースを介して行います。

読み取り/書き込みロック関連のプローブは、表 31–2 のとおりです。それぞれの `arg0` には、`rwlock_t` 構造体または `pthread_rwlock_t` 構造体 (型はすべて同じ) のポインタが入ります。これは適応型ロックを表します。`arg1` には、行われた操作が書き込みであるかどうかを示すブール値が入ります。

表 31–2 読み取り/書き込みロックプローブ

`rw-acquire`	読み取り/書き込みロックの獲得後すぐに起動する保持イベントプローブ。
`rw-block`	ロックの獲得中、スレッドがブロックする前に起動する競合イベントプローブ。有効にすると、`rw-acquire` プローブまたは `rw-error` プローブが `rw-block` に続いて起動します。
`rw-release`	読み取り/書き込みロックの解放後すぐに起動する保持イベントプローブ。
`rw-error`	読み取り/書き込みロック処理でエラーが発生したときに起動するエラーイベントプローブ。`arg1` には、発生したエラーの `errno` 値が入ります。

安定性

以下の表に、`plockstat` プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	発展中	発展中	ISA

◆ ◆ ◆ 第 32 章

fasttrap プロバイダ

fasttrap プロバイダでは、事前にプログラミングされた特定のユーザープロセスの位置でトレースできます。fasttrap プロバイダは、ほかの大部分の DTrace プロバイダと違って、システムアクティビティーをトレースするように設計されていません。このプロバイダは、DTrace コンシューマが fasttrap プローブを有効にすることで、DTrace フレームワークに情報を注入することを目的としています。

プローブ

fasttrap プロバイダは、ユーザーレベルのプロセスがカーネルに対して DTrace 呼び出しを行なったときに起動するプローブ、fasttrap:::fasttrap を有効にします。このプローブを有効にする DTrace 呼び出しは、現時点では正式に公開されていません。

安定性

以下の表に、fasttrap プロバイダの安定性を DTrace の安定性機構に従って示します。安定性機構の詳細については、第 39 章「安定性」を参照してください。

要素	名前の安定性	データの安定性	依存クラス
プロバイダ	発展中	発展中	ISA
モジュール	非公開	非公開	不明
機能	非公開	非公開	不明
名前	発展中	発展中	ISA
引数	発展中	発展中	ISA

◆ ◆ ◆ 第 33 章

33 ユーザープロセスのトレース

DTrace は、ユーザープロセスの動作を把握するために役立つ非常に強力なツールです。デバッグ時やパフォーマンスの問題の解析時に、あるいはただ複雑なアプリケーションの動作を理解したい場合にも、DTrace は非常に有益です。この章では、ユーザープロセスアクティビティに関する DTrace 機能に注目し、例を挙げながらその使用法について説明します。

サブルーチン copyin() と copyinstr()

DTrace でのプロセスの扱い方は、従来のデバッガや監視ツールとは若干異なっています。従来のツールは、プロセスのスコープ内で実行されます。このため、プログラム変数のポインタの間接参照をユーザー側で行うことができます。これに対して、DTrace プローブは、Solaris カーネル内で実行されます。プロセス内で (プロセスの一部として) 実行されるわけではありません。プロセスデータにアクセスする場合、プローブは、サブルーチン `copyin()` または `copyinstr()` を使って、ユーザープロセスデータをカーネルのアドレス空間にコピーする必要があります。

たとえば、次のような `write(2)` システムコールがあるとします。

```
ssize_t write(int fd, const void *buf, size_t nbytes);
```

では、`write(2)` システムコールに渡される文字列の内容を出力するには、どうしたらよいでしょうか。以下は、正しくない D プログラムの例です。

```
syscall::write:entry
{
    printf("%s", stringof(arg1)); /* incorrect use of arg1 */
}
```

このスクリプトを実行すると、次のようなエラーメッセージが出力されます。

```
dtrace: error on enabled probe ID 1 (ID 37: syscall::write:entry): \
    invalid address (0x10038a000) in action #1
```

arg1 変数には、*buf* パラメータの値として、システムコールの実行プロセス内のメモリーを参照するアドレスが入ります。このアドレスの文字列を読み取るには、copyinstr() サブルーチンを使用し、その結果を printf() アクションで記録します。

```
syscall::write:entry
{
    printf("%s", copyinstr(arg1)); /* correct use of arg1 */
```

このスクリプトの出力から、write(2) システムコールに渡されるすべての文字列を確認できます。ただし、以下のような異常な出力が得られる場合もあります。

```
  0     37                        write:entry madaï¿½ï¿½ï¿½
```

copyinstr() サブルーチンは、入力引数 (NULL で終わっている ASCII 文字列のユーザーアドレス) に対して機能します。しかし、write(2) システムコールに渡されるバッファーが、ASCII 文字列ではなくバイナリデータを参照していることがあります。呼び出し側が意図した文字列だけを出力するには、copyin() サブルーチンを使用します。このサブルーチンの第 2 引数にサイズを指定します。

```
syscall::write:entry
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

stringof 演算子がないと、DTrace は、copyin() を使って取得したユーザーデータを正しく文字列に変換できません。しかし、copyinstr() を使用する場合は、stringof は不要になります。これは、copyinstr が常に string 型を返すからです。

エラーの回避

サブルーチン copyin() や copyinstr() は、過去にアクセスしたことのないユーザーアドレスを読み取ることができません。アドレスが含まれているページが、過去にアクセスされてフォルトインされたことがないと、たとえそのアドレスが有効であっても、エラーが起きる可能性があります。次の例で考えてみてください。

```
# 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
```

この出力例では、アプリケーションが正常に機能していて、arg0 に指定されたアドレスも有効ですが、対応するプロセスがまだアクセスしたことがないページが参照

されています。この問題を解決するには、カーネルまたはアプリケーションでデータが使用されるのを待ってからトレースを開始する必要があります。たとえば、システムコールが復帰して、copyinstr() が適用されるまで待ちます。次の例を参照してください。

```
# 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
```

dtrace(1M) の干渉の排除

write(2) システムコールの呼び出しをすべてトレースすると、結果が際限なく出力されます。write() の呼び出しのたびに、dtrace(1M) コマンドが出力表示のために write() を呼び出します。このフィードバックループは、dtrace コマンドの干渉によって望ましくないデータが出力される一例です。このような不要なデータのトレースを防ぐには、次の述語を使用します。

```
syscall::write:entry
/pid != $pid/
{
    printf("%s", stringof(copyin(arg1, arg2)));
}
```

$pid マクロ変数は、プローブを有効にしたプロセスのプロセス ID に展開されます。pid 変数には、プローブが起動した CPU 上でスレッドを実行していたプロセスのプロセス ID が入ります。このため、述語 /pid != $pid/ を使用すれば、このスクリプト自体の実行に関連したイベントをトレースしないようにできます。

syscall プロバイダ

syscall プロバイダでは、すべてのシステムコールの開始 (entry) と終了 (return) をトレースできます。プロセスの動作を把握するときは、まずシステムコールに注目します。特に、そのプロセスがカーネル内で長時間実行されていたり、長時間ブロックされているように見える場合は、そうすることをお勧めします。プロセスがどこで時間を消費しているかを確認するには、prstat(1M) コマンドを使用します。

```
$ 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
```

この例では、あるプロセスが多くのシステム時間を消費しています。この動作の原因の 1 つとして考えられるのは、このプロセスが大量のシステムコールを実行して

いるのではないかということです。コマンド行から、一番頻繁に呼び出されているシステムコールを調べる単純な D プログラムを実行してみましょう。

```
# 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
```

出力結果から、一番頻繁に呼び出されているシステムコールがわかります。この例では、write(2) システムコールが該当します。syscall プロバイダを使って、すべての write() システムコールの呼び出し元について詳しく調べることができます。

```
# 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
```

このプロセスは、比較的少量のデータで、多数の write() システムコールを実行しています。このプロセスの場合、このアンバランスがパフォーマンスの問題の原因になっていると考えられます。これで、システムコールの動作について調査する一般的な方法がわかりました。

ustack() アクション

問題をより詳しく調査するときは、特定のプローブが起動したときのプロセススレッドのスタックをトレースするとよいでしょう。ustack() アクションは、ユーザースレッドのスタックをトレースします。たとえば、多数のファイルを開くプロセスがあるとします。このプロセスは、ときどき open(2) システムコールに失敗します。この場合、問題のある open() を実行するコードパスを探すには、ustack() アクションを使用します。

```
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();
}
```

このスクリプトでは、マクロ変数 $1 も使用されています。このマクロ変数には、dtrace(1M) コマンド行で指定された最初のオペランドの値が入ります。

```
# 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
```

ustack() アクションによって、スタックのプログラムカウンタ (PC) の値が記録されます。すると、dtrace(1M) により、プロセスのシンボルテーブルが検索され、PC 値がシンボル名に解決されます。dtrace が PC 値をシンボル名に解決できない場合は、その値は 16 進整数として出力されます。

ustack() データの出力書式が設定される前にプロセスが終了 (強制終了も含む) すると、dtrace が、スタックトレース内の PC 値をシンボル名に変換できない場合があります。この場合、これらの値は 16 進整数で表示されます。この制限を回避するに

は、-dtrace コマンドの -c オプションや p オプションを使って、対象プロセスを指定します。これらのオプションやその他のオプションの詳細については、第 14 章「dtrace(1M) ユーティリティー」を参照してください。プロセス ID やコマンドがあらかじめわかっていない場合は、次のような D プログラムで制限を回避できます。

```
/*
 * 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;
}
```

このスクリプトは、プロセス内のスレッドに ustack() アクションが適用されている場合、プロセスを、その終了直前に停止します。この方法を利用すれば、dtrace コマンドで PC 値をシンボル名に解決できます。stop_pids[pid] の値は、動的変数の消去が完了したら、ゼロになります。prun(1) コマンドを使って、停止したプロセスが実行再開するように設定してください。そうしないと、システム上の停止プロセスが蓄積されてしまいます。

uregs[] 配列

uregs[] 配列を使用すると、個々のユーザーレジスタにアクセスできます。以下の表に、uregs[] 配列のインデックスを、Solaris でサポートされているシステムアーキテクチャ別に示します。

表 33–1 SPARC uregs[] 定数

定数	レジスタ
R_G0..R_G7	%g0..%g7 汎用レジスタ
R_O0..R_O7	%o0..%o7 アウトレジスタ

表 33–1 SPARC uregs[] 定数 *(続き)*

定数	レジスタ
R_L0..R_L7	%l0..%l7 ローカルレジスタ
R_I0..R_I7	%i0..%i7 インレジスタ
R_CCR	%ccr 条件コードレジスタ
R_PC	%pc プログラムカウンタ
R_NPC	%npc 次のプログラムカウンタ
R_Y	%y 乗算/除算レジスタ
R_ASI	%asi アドレス空間識別レジスタ
R_FPRS	%fprs 浮動小数点レジスタの状態

表 33–2 x86 uregs[] 定数

定数	レジスタ
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
R_CS	%cs

表 33–2 x86 uregs[] 定数 (続き)

定数	レジスタ
R_ERR	%err
R_EFL	%efl
R_UESP	%uesp
R_SS	%ss

AMD64 プラットフォームの uregs 配列には、x86 プラットフォームの内容に加えて、以下の表の要素が含まれます。

表 33–3 amd64 uregs[] 定数

定数	レジスタ
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

すべてのプラットフォームで使用できる別名は、以下の表のとおりです。

表 33–4 共通の uregs[] 定数

定数	レジスタ
R_PC	プログラムカウンタレジスタ
R_SP	スタックポインタレジスタ
R_R0	最初の復帰コード
R_R1	2 番目の復帰コード

pid プロバイダ

pid プロバイダでは、プロセス内の任意の命令をトレースできます。大半のプロバイダとは異なり、pid プローブは、D プログラム内のプローブ記述で、オンデマンドで作成されます。そのため、ユーザー自身で pid プローブを有効にしないかぎり、dtrace -l を実行しても何も出力されません。

ユーザー関数境界のトレース

pid プロバイダのもっとも単純な操作モードは、fbt プロバイダにとってのユーザー空間に似ています。以下は、ある関数の開始 (entry) と終了 (return) をすべてトレースするプログラム例です。$1 マクロ変数 (コマンド行の最初のオペランド) には、トレース対象のプロセスのプロセス ID が入ります。$2 マクロ変数 (コマンド行の 2 番目のオペランド) には、すべての関数呼び出しをトレースする関数の名前が入ります。

例 33–1 userfunc.d: ユーザー関数の開始 (entry) と終了 (return) のトレース

```
pid$1::$2:entry
{
    self->trace = 1;
}

pid$1::$2:return
/self->trace/
{
    self->trace = 0;
}

pid$1:::entry,
pid$1:::return
/self->trace/
{
}
```

このスクリプトを userfunc.d という名前のファイルに保存し、chmod で実行可能ファイルに変更します。このスクリプトからは、次のような結果が出力されます。

```
# ./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
...
```

pid プロバイダは、すでに実行中のプロセスに対してしか使用できません。$target マクロ変数 (第 15 章「スクリプトの作成」を参照) と、dtrace コマンドの -c オプションと -p オプションを使って、プロセスを作成し、取り込みます。さらに、DTrace を使って、これらのプロセスを計測します。たとえば、次の D スクリプトでは、特定の従属プロセスによって実行される libc の関数呼び出しの内訳がわかります。

```
pid$target:libc.so::entry
{
    @[probefunc] = count();
}
```

このスクリプトを libc.d という名前のファイルに保存し、次のコマンドを実行すると、date(1) コマンドによって実行されるこの種の呼び出しの内訳がわかります。

```
# 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
```

任意の命令のトレース

pid プロバイダでは、任意のユーザー関数内の任意の命令をトレースできます。pid プロバイダは、必要に応じて、関数内の各命令に対して 1 つずつプローブを作成します。各プローブの名前は、関数内の対応する命令のオフセット (16 進整数) になります。たとえば、PID 123 のプロセス内にあるモジュール bar.so の関数 foo で、オフセット 0x1c にある命令に関連したプローブを有効にしたい場合は、次のコマンドを使用します。

```
# dtrace -n pid123:bar.so:foo:1c
```

関数 foo 内のプローブを、各命令用のプローブも含めてすべて有効にするには、次のコマンドを使用します。

```
# dtrace -n pid123:bar.so:foo:
```

このコマンドは、ユーザーアプリケーションのデバッグと解析にたいへん役立ちます。発生頻度の低いエラーは、再現性が低いため、簡単にはデバッグできません。また、エラーの発生後、ようやく問題を突き止めたときには、すでにコードパスを再現できなくなっている場合もあります。以下の例のように、pid プロバイダと投機トレース (第 13 章「投機トレース」を参照) を組み合わせて、関数内のすべての命令を個別にトレースすると、この問題を解決できます。

例 33–2 errorpath.d: ユーザー関数呼び出しのエラーパスをトレース

```
pid$1::$2:entry
{
    self->spec = speculation();
    speculate(self->spec);
    printf("%x %x %x %x %x", arg0, arg1, arg2, arg3, arg4);
}
```

例 33–2 errorpath.d: ユーザー関数呼び出しのエラーパスをトレース (続き)

```
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;
}
```

errorpath.d を実行すると、次のような結果が出力されます。

```
# ./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
```

◆ ◆ ◆ 第 34 章

ユーザーアプリケーション向けの静的に定義されたトレース

DTrace には、pid プロバイダの機能を拡張するため、アプリケーションコード内にカスタムプローブを定義する機能があります。この機能は、ユーザーアプリケーション開発者向けです。これらのカスタムプローブは、静的プローブです。無効にすると、ほとんどオーバーヘッドはかかりません。有効にするときは、ほかの DTrace プローブと同じように動的に有効にします。こうした静的プローブを使用すれば、アプリケーションの実装に関する知識があるかどうかを問わず、DTrace ユーザーにアプリケーションのセマンティクスを示すことができます。この章では、ユーザーアプリケーション内に静的プローブを定義する方法と、DTrace を使ってこうしたプローブをユーザープロセス内で有効にする方法について説明します。

プローブポイントの選択

DTrace では、開発者は、アプリケーションコード内 (完成したアプリケーションと共有ライブラリを含む) に静的プローブポイントを埋め込むことができます。それが開発環境であろうが本稼働環境であろうが、そのアプリケーションやライブラリを実行しているときはいつでも、これらのプローブを有効にすることができます。プローブを定義するときは、DTrace ユーザーコミュニティにとって、その意味が理解しやすいかどうかを考慮する必要があります。たとえば、要求の送付元クライアントと通信する Web サーバーに query-receive プローブ、その要求に応答する Web サーバーには query-respond プローブを定義するとします。ほとんどの DTrace ユーザーは、これらのプローブの意味を難なく理解できます。これらのプローブは、アプリケーションの下位の実装詳細レベルではなく、最上位の抽象レベルに対応しています。DTrace ユーザーは、これらのプローブを使って、要求の時間的分布を調べることができます。query-receive プローブの引数が URL 要求文字列である場合は、このプローブと io プロバイダを組み合わせて、ディスク入出力の大部分を生成している要求を特定できます。

プローブ名とプローブの位置を選択するときは、抽象化の安定性についても考慮する必要があります。アプリケーションの将来のリリースで実装が変更されても、引き続き使用できるプローブかどうか。どのシステムアーキテクチャにも通用するプ

ローブか。あるいは、特定の命令セットに固有のプローブか。この章では、こうした判断をもとに静的トレースの定義を行なっていく方法を詳しく説明します。

アプリケーションへのプローブの追加

ライブラリや実行可能ファイルの DTrace プローブは、対応するアプリケーションバイナリの ELF セクションに定義されます。この節では、プローブを定義し、アプリケーションのソースコードに追加することにより、アプリケーションの構築プロセスに DTrace プローブ定義を追加する方法について説明します。

プロバイダとプローブの定義

`.d` ソースファイルに DTrace プローブを定義します。このソースファイルは、あとでアプリケーションのコンパイルとリンク時に使用されます。まず、適切なユーザーアプリケーションプロバイダの名前を選択します。アプリケーションコードを実行している各プロセスのプロセス ID に、選択したプロバイダ名が追加されます。たとえば、プロセス ID 1203 で実行されている Web サーバー用としてプロバイダ名 `myserv` を選択した場合、このプロセスに対応する DTrace プロバイダ名は `myserv1203` になります。`.d` ソースファイルに、次のようなプロバイダ定義を追加します。

```
provider myserv {
    ...
};
```

次に、各プローブとその引数の定義を追加します。次の例では、373 ページの「プローブポイントの選択」で説明した 2 つのプローブを定義しています。最初のプローブは、`string` 型の引数を 2 つとります。2 番目のプローブは引数をとりません。D コンパイラは、プローブ名に含まれる 2 つの連続した下線 (`--`) を単一のハイフン (`-`) に変換します。

```
provider myserv {
    probe query__receive(string, string);
    probe query__respond();
};
```

プロバイダ定義には安定性属性を追加して、プローブの使用者がアプリケーションの将来のバージョンでの変更可能性について判断できるようにしてください。DTrace の安定性属性の詳細については、第 39 章「安定性」を参照してください。以下は、安定性属性の定義例です。

例 **34–1** `myserv.d`: 静的に定義されたアプリケーションプローブ

```
#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
```

例 34–1 myserv.d: 静的に定義されたアプリケーションプローブ (続き)

```
#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();
};
```

注 – ユーザーが追加したプローブの整数でない引数を D スクリプト内で使用する場合、copyin() および copyinstr() 関数を使ってそれらの引数を取得する必要があります。詳細については、第 33 章「ユーザープロセスのトレース」を参照してください。

アプリケーションコードへのプローブの追加

.d ファイルを使ってプローブを定義できたら、ソースコードに、プローブがトリガーされる位置の情報を追加する必要があります。次のような C アプリケーションソースコードがあるとします。

```
void
main_look(void)
{
    ...
    query = wait_for_new_query();
    process_query(query)
    ...
}
```

プローブの位置の情報を追加するには、<sys/sdt.h>に定義されている DTRACE_PROBE() マクロの参照を追加します。次の例を参照してください。

```
#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)
```

```
    ...
}
```

マクロ名 DTRACE_PROBE2 の接尾辞 2 は、プローブに渡される引数の数を表します。プローブマクロの最初の 2 つの引数は、プロバイダ名とプローブ名です。これらはそれぞれ、D プロバイダ定義とプローブ定義に一致していなければなりません。残りのマクロ引数は、プローブの起動時に DTrace arg0..9 変数に割り当てられる引数です。アプリケーションソースコード内で同じプロバイダ名やプローブ名を繰り返し参照できます。ソースコード内で同じプローブが何回も参照されている場合、そのいずれかがプローブを起動することになります。

プローブを含むアプリケーションの構築

アプリケーションの構築プロセスに、DTrace プロバイダとプローブの定義を追加する必要があります。通常の構築プロセスでは、各ソースファイルがコンパイルされ、対応するオブジェクトファイルが生成されます。次に、このコンパイルされたオブジェクトファイルが互いにリンクされることにより、最終的なアプリケーションバイナリが完成します。次の例を参照してください。

```
cc -c src1.c
cc -c src2.c
...
cc -o myserv src1.o src2.o ...
```

アプリケーションに DTrace プローブ定義を追加するには、構築プロセスに適切な Makefile 規則を追加して、dtrace コマンドを実行します。次の例を参照してください。

```
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 ...
```

この dtrace コマンドは、コンパイラコマンドによって生成されたオブジェクトファイルに後処理を施すことにより、myserv.d とその他のオブジェクトファイルから myserv.o という名前のオブジェクトファイルを生成します。プロバイダ定義とプローブ定義をユーザーアプリケーションにリンクするには、dtrace -G オプションを使用します。32 ビットアプリケーションバイナリを構築するには、-32 オプションを使用します。64 ビットアプリケーションバイナリを構築するには、-64 オプションを使用します。

◆ ◆ ◆ 第 35 章

セキュリティー

この章では、システム管理者が特定のユーザーやプロセスに DTrace へのアクセス権を付与するために使用する権限について説明します。DTrace では、ユーザーレベルの関数、システムコール、カーネル関数など、あらゆる面に注目して、システムを把握できます。プログラムの状態を変更してしまうような、強力なアクションも用意されています。ユーザーに別のユーザーの個人用ファイルへのアクセス権を付与するのが不適切であるように、システム管理者は、すべてのユーザーにすべての DTrace 機能へのアクセス権を付与すべきではありません。デフォルトでは、DTrace を使用できるのはスーパーユーザーだけです。その他のユーザーに DTrace の制御付き使用を許可するときは、「最小権限」の機能を使用します。

特権

Solaris の最小権限機能を利用して、管理者は、特定の Solaris ユーザーに特定の権限を付与できます。ユーザーのログイン時に権限を付与したい場合は、`/etc/user_attr` ファイルに次の 1 行を挿入します。

user-name`::::defaultpriv=basic,`*privilege*

実行中のプロセスに権限を追加したい場合は、`ppriv(1)` コマンドを使用します。

`# ppriv -s A+`*privilege process-ID*

DTrace 機能へのユーザーアクセスは、`dtrace_proc`、`dtrace_user`、`dtrace_kernel` の 3 つの権限で制御します。権限によって、使用を許可される DTrace プロバイダ、アクション、および変数が異なります。また、各権限は、DTrace の特定の用途に対応しています。権限モードの詳細については、次節以降で説明します。システム管理者は、個々のユーザーのニーズと、個々の権限モードによる可視性やパフォーマンスへの影響を慎重に比較考慮する必要があります。ユーザーが DTrace の機能を使用するには、3 つの DTrace 権限のうち最低 1 つが必要です。

DTrace の権限付き使用

3 つの DTrace 権限のうちのどれかを持つユーザーは、dtrace プロバイダ (第 17 章「dtrace プロバイダ」を参照) から提供されるプローブを有効にすることができます。また、次のアクションと変数を使用できます。

プロバイダ	dtrace		
アクション	exit	printf	tracemem
	discard	speculate	
	printa	trace	
変数	args	probemod	this
	epid	probename	timestamp
	id	probeprov	vtimestamp
	probefunc	self	
アドレス空間	なし		

dtrace_proc 権限

dtrace_proc 権限が付与されている場合は、プロセスレベルのトレースで、fasttrap プロバイダを使用できます。次のアクションおよび変数も使用できます。

アクション	copyin	copyout	stop
	copyinstr	raise	ustack
変数	execname	pid	uregs
アドレス空間	ユーザー		

この権限を持っていても、Solaris カーネルデータ構造や、ユーザーがアクセス権を持っていないプロセスに対しては、可視性が付与されません。

この権限を持っているユーザーは、自分が所有しているプロセス内に限って、プローブを作成したり有効にしたりできます。この権限に加えて proc_owner 権限を持っていれば、どのプロセス内でも、プローブを作成したり有効にしたりできます。dtrace_proc 権限は、ユーザープロセスのデバッグやパフォーマンス解析に関心のあるユーザー向けです。この権限は、新しいアプリケーションの開発者や、本稼働環境でアプリケーションのパフォーマンス改善を担当するエンジニアに最適です。

注 – dtrace_proc 権限と proc_owner 権限の両方を持つユーザーは、任意のプロセスの任意の pid プローブを有効化することができます。ただし、プロセス内にプローブを作成するためには、そのプロセスの権限セットが、このユーザー自身の権限セットのサブセットである必要があります。詳細については、最小権限に関する文書を参照してください。

dtrace_proc 権限では、ユーザーがアクセス権を持っているプロセスのパフォーマンスだけに悪影響を及ぼす可能性がある DTrace アクセスが許可されます。計測されるプロセスは、システムリソースに対して、より多くの負荷をかけます。このため、システム全体のパフォーマンスにも多少影響を及ぼします。このように全体の負荷が増加する点を別にすると、この権限は、トレース対象外のプロセスのパフォーマンスに影響を及ぼすような計測機能を許可しません。この権限は、別のプロセスやカーネル自体に対する可視性をユーザーに与えないため、所有しているプロセスの内部動作を把握する必要があるすべてのユーザーに付与することを推奨します。

dtrace_user 権限

dtrace_user 権限では、制限付きで profile プロバイダと syscall プロバイダの使用を許可します。また、次のアクションと変数の使用を許可します。

プロバイダ	プロファイル	syscall	fasttrap
アクション	copyin	copyout	stop
	copyinstr	raise	ustack
変数	execname	pid	uregs
アドレス空間	ユーザー		

dtrace_user 権限は、ユーザーがすでにアクセス権を持っているプロセスだけに対して可視性を提供します。カーネルの状態やアクティビティに対する可視性は提供しません。この権限を持つユーザーは、syscall プロバイダを有効にすることができます。しかし、有効化されたプローブは、このユーザーがアクセス権を持っているプロセス内でしか起動しません。profile プロバイダの場合も同様です。このプロバイダも有効にすることができますが、有効化されたプローブは、このユーザーがアクセス権を持っているプロセス内でしか起動しません。したがって、このプローブは Solaris カーネル内では絶対に起動しません。

この権限は、特定のプロセスだけに対して可視性を提供する一方で、システム全体のパフォーマンスに影響を及ぼすような計測機能の使用を許可します。syscall プロバイダは、すべてのプロセスのすべてのシステムコールのパフォーマンスに若干の

影響を及ぼします。profile プロバイダは、リアルタイムタイマーと同様に、決まった時間間隔で実行され、システム全体のパフォーマンスに影響を及ぼします。これらのパフォーマンスの低減は、いずれも顕著なものではなく、システムの処理の進行に重大な影響を及ぼすことはありません。しかし、システム管理者は、この権限をユーザーに付与することの意味を十分に考慮する必要があります。syscall および profile プロバイダのパフォーマンスへの影響については、第 21 章「syscall プロバイダ」と第 19 章「profile プロバイダ」を参照してください。

dtrace_kernel 権限

dtrace_kernel 権限は、ユーザーが所有していないプロセス上の pid プロバイダと fasttrap プロバイダを除く、すべてのプロバイダの使用を許可します。この権限は、カーネル破壊アクション (breakpoint()、panic()、chill()) を除く、すべてのアクションおよび変数の使用も許可します。この権限は、カーネルとユーザーの状態すべてに対する可視性を提供します。dtrace_user 権限によって有効になる機能は、dtrace_kernel で有効になる機能の厳密なサブセットです。

プロバイダ	制限範囲内ですべて
アクション	すべて (破壊アクションを除く)
変数	すべて
アドレス空間	ユーザー　　カーネル

スーパーユーザーの権限

すべての権限を持っているユーザーは、すべてのプロバイダとすべてのアクションを使用できます。使用できるアクションには、ほかのクラスのユーザーには許可されていない、カーネル破壊アクションも含まれています。

プロバイダ	すべて
アクション	すべて (破壊アクションを含む)
変数	すべて
アドレス空間	ユーザー　　カーネル

◆ ◆ ◆ 第 36 章

匿名トレース

この章では、「匿名トレース」と呼ばれる、DTrace コンシューマに関連付けられていないトレースについて説明します。匿名トレースは、DTrace コンシューマプロセスを実行できない場合に使用します。匿名トレースは、通常、デバイスドライバ開発者が、システムのブート時のアクティビティをデバッグしたりトレースしたりするために使用します。対話形式でトレース可能なデータは、匿名でもトレースできます。ただし、匿名有効化を作成できるのはスーパーユーザーだけです。また、複数の匿名有効化が同時に存在することはありません。

匿名有効化

匿名有効化を作成するには、dtrace(1M) を -A オプション付きで呼び出して、必要なプローブ、述語、アクション、およびオプションを指定します。dtrace は、dtrace(7D) ドライバの構成ファイル (通常は /kernel/drv/dtrace.conf) に、要求内容に対応した一連のドライバプロパティを追加します。これらのプロパティは、ロード後、dtrace(7D) ドライバによって読み取られます。このドライバは、指定されたアクション関連の指定されたプローブを有効にし、「匿名状態」を作成して、新しい有効化に関連付けます。通常、dtrace(7D) ドライバは、DTrace プロバイダとして機能するほかのドライバと同じように、オンデマンドでロードされます。ブート中にトレースを行うためには、なるべく早い段階で dtrace(7D) ドライバをロードする必要があります。dtrace は、必要な DTrace プロバイダと自身 (dtrace(7D)) の /etc/system (system(4) のマニュアルページを参照) に、必要な forceload 文を追加します。

その後、システムがブートすると、dtrace(7D) から、構成ファイルが正常に処理されたことを示すメッセージが発行されます。

匿名有効化には、バッファーサイズ、動的変数のサイズ、投機のサイズ、投機の数など、あらゆるオプションを設定できます。

匿名有効化を削除するには、プローブ記述を指定しないで、dtrace -A を実行します。

匿名状態を要求する

マシンが完全にブートしたら、任意の匿名状態を要求できるようになります。このためには、-a オプションを指定して dtrace を実行します。デフォルトでは、-a は、匿名状態を要求し、既存のデータを処理し、実行を続行することを表します。匿名状態を消費して終了する場合は、-e オプションを追加します。

カーネルの匿名状態がいったん消費されてしまうと元に戻すことはできませんが、カーネル内バッファーは再利用されます。匿名トレース状態が存在しないのに、この状態を要求した場合、dtrace から次のようなメッセージが返されます。

```
dtrace: could not enable tracing: No anonymous tracing state
```

欠落やエラーが発生した場合、dtrace は、匿名状態が要求された時点で適切なメッセージを返します。欠落やエラーを知らせるメッセージは、匿名状態のときも非匿名状態のときも同じです。

匿名トレースの例

以下の例では、iprb(7D) モジュール内の各プローブを DTrace で匿名有効化します。

```
# 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
```

リブート後、コンソールに、指定されたプローブを有効化していることを示す dtrace(7D) からのメッセージが出力されます。

```
  ...
  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.
  ...
```

マシンのリブートが完了したら、dtrace -a コマンドを実行して、匿名状態を消費できます。

```
# 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
...
```

以下の例では、`iprbattach()` から呼び出された関数だけに注目します。エディタで以下のスクリプトを入力し、`iprb.d` という名前のファイルに保存してください。

```
fbt::iprbattach:entry
{
    self->trace = 1;
}

fbt:::
/self->trace/
{}

fbt::iprbattach:return
{
    self->trace = 0;
}
```

次のコマンドを実行します。すると、ドライバ構成ファイルの以前の設定が消去され、新しい匿名トレース要求がインストールされます。そしてリブートします。

```
# 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
```

リブート後、コンソールに `dtrace(7D)` からのメッセージが出力されます。出力される有効化の情報は、前回とは若干異なっています。

```
...
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.
...
```

マシンのブートが完了したら、dtrace に -a オプションと -e オプションを付けて実行します。こうすると、匿名データを消費したあとに終了します。

```
# 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
...
```

◆ ◆ ◆ 第 37 章

37 事後トレース

この章では、DTrace コンシューマのカーネル内データを「事後」に抽出し処理する、DTrace 機能について説明します。たとえばシステムがクラッシュした場合、それまでに DTrace で記録された情報が、障害の根本原因を探る重要な手がかりになる可能性があります。システムのクラッシュダンプから DTrace データを抽出し、処理したあと、このデータを基に、システムの致命的な障害について調べることができます。この DTrace の事後機能とリングバッファーポリシー (第 11 章「バッファーとバッファリング」を参照) を組み合わせると、DTrace は、オペレーティングシステム内で、飛行機の「ブラックボックス」(フライトデータレコーダー) と同じような役割を果たします。

クラッシュダンプから DTrace データを抽出するには、まず、そのクラッシュダンプに対して Solaris モジューラデバッガ (mdb(1)) を実行します。すると、DTrace 機能が格納されている MDB モジュールが自動的にロードされます。MDB の詳細については、『『Solaris モジューラデバッガ』』を参照してください。

DTrace コンシューマの表示

DTrace コンシューマから DTrace データを抽出するには、まず ::dtrace_state MDB dcmd を実行して、抽出元の DTrace コンシューマを特定する必要があります。

```
> ::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
```

このコマンドを実行すると、DTrace の状態構造が表形式で出力されます。表の各行には、次の情報が含まれています。

- 状態構造のアドレス

- dtrace(7D) デバイスのマイナー番号
- DTrace コンシューマのプロセス構造のアドレス
- DTrace コンシューマ名 (匿名コンシューマの場合は <anonymous>)
- dtrace(7D) オープンデバイスのファイル構造の名前

特定の DTrace コンシューマの詳細情報が必要な場合は、::ps dcmd に、そのプロセス構造のアドレスを指定します。

```
> d71377f0::ps
S    PID   PPID   PGID    SID    UID      FLAGS     ADDR NAME
R 100647 100642 100647 100638      0 0x00004008 d71377f0 dtrace
```

トレースデータの表示

コンシューマを特定したら、::dtrace dcmd に状態構造のアドレスを指定して、まだ消費されていないバッファーのデータを検出します。以下は、trace(execname) アクションが関連付けられている syscall:::entry の匿名有効化に対して ::dtrace dcmd を実行したときの出力例です。

```
> ::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
...
```

::dtrace dcmd は、dtrace(1M) と同じようにしてエラーを処理します。コンシューマの実行中に、欠落、エラー、投機欠落などが発生した場合、::dtrace は、dtrace(1M) メッセージと同様のメッセージを発行します。

::dtrace は、常に CPU 内の古いイベントから順に出力します。CPU バッファー自体は、番号順に出力されます。複数の異なる CPU 上のイベントに番号を付ける必要がある場合は、timestamp 変数をトレースします。

特定の CPU のデータだけを出力したい場合は、::dtrace に -c オプションを指定します。

```
> 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
...
```

::dtrace は、カーネル内の D トレースデータしか処理しません。カーネルから消費され、dtrace(1M) などによって処理されたデータを、再度 ::dtrace で処理することはできません。障害発生時にもデータを最大限に確保するには、リングバッファーポリシーを使用します。バッファーポリシーの詳細については、第 11 章「バッファーとバッファリング」を参照してください。

以下は、非常に小さい (16K) リングバッファーを作成し、すべてのシステムコールとその呼び出し元プロセスを記録する例です。

```
# dtrace -P syscall'{trace(curpsinfo->pr_psargs)}' -b 16k -x bufpolicy=ring
dtrace: description 'syscall:::entry' matched 214 probes
```

このコマンドを実行したときのクラッシュダンプは、以下のようになります。

```
> ::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
>
```

CPU 1 の最新のレコードに、`mdb -kw` プロセスによる一連の `write(2)` システムコールが含まれている点に注目してください。この結果がシステム障害の原因に関係して

いると推測されます。というのは、mdb(1) に -k オプションと -w オプションを指定して実行すると、実行中のカーネルのデータやテキストをユーザーが変更できるからです。この例では、DTrace のデータから、障害の原因そのものではないにしても、それを探る道筋のようなものを見出すことができます。

◆ ◆ ◆ 第 38 章

38 性能に関する考慮事項

DTrace の使用は、システム内での作業量の増加につながります。そのため、DTrace を有効にすると、必ずシステムパフォーマンスに何らかの影響があります。通常、この影響はごくわずかです。しかし、処理コストの高い有効化によって多くのプローブを有効化した場合は、無視できない影響を与える可能性があります。この章では、DTrace のパフォーマンスへの影響を最小限に抑えるテクニックを紹介します。

有効にするプローブの制限

動的計測機能の技術により、DTrace では、カーネルや任意のユーザープロセスについて、広範囲のトレース対象が提供されています。この対象範囲の広さによって、システムの動作を調べる能力は飛躍的に向上しました。しかし、一方では、プローブの影響が非常に大きくなったと言うこともできます。何万個というレベルで大量のプローブを有効にした場合、システムに多大な影響が出るのは明らかです。したがって、問題解決に必要最低限のプローブだけを有効にしてください。たとえば、より小規模な有効化で問題が解決できるならば、すべての FBT プローブを有効にすることは避けてください。特定のモジュールや、特定の関数に注目するだけで、必要な答えが得られる可能性があります。

`pid` プロバイダを使用するときは、特に注意が必要です。`pid` プロバイダは、すべての命令を計測する可能性があります。その場合、アプリケーション内の大量のプローブが有効になり、ターゲットプロセスの処理速度が異常に遅くなってしまいます。

DTrace は、大量のプローブを有効にしなければ問題への答えが得られないような状況でも使用できます。大量のプローブを有効にすると、システムの処理速度が低減しますが、そのせいで致命的な障害が発生するわけではありません。必要な場合は、大量のプローブを有効にしても、まったく問題はありません。

集積体の使用

第 9 章「集積体」で説明したように、DTrace の集積体を使用すると、データをスケーラブルに集積できます。連想配列にも、集積体とよく似た働きがあります。しかし、大域的な汎用変数としての性質上、集積体のようなリニアなスケーラビリティを提供できません。可能なかぎり、連想配列ではなく集積体を使用してください。以下は、推奨しない例です。

```
syscall::entry
{
    totals[execname]++;
}

syscall::rexit:entry
{
    printf("%40s %d\n", execname, totals[execname]);
    totals[execname] = 0;
}
```

以下が、望ましい例です。

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

END
{
    printa("%40s %@d\n", @totals);
}
```

キャッシュ可能な述語の使用

DTrace では、述語を使って、計測結果から不要なデータを排除できます。指定された条件が満たされた場合だけ、データのトレースが行われます。多数のプローブを有効にするときは、通常、特定のスレッド (複数可) を識別するような形式の述語を使用します。該当する例として、`/self->traceme/` や `/pid == 12345/` を挙げることができます。これらの述語の多くは、ほとんどのプローブのほとんどのスレッドに対して偽を返します。しかし、プローブの数があまりにも多いと、この評価自体にかなりのコストがかかるようになります。このコストを下げるため、DTrace は、スレッド固有変数 (`/self->traceme/` など) や不変変数 (`/pid == 12345/` など) だけが含まれている述語の評価をキャッシュに格納します。キャッシュに入った述語は、キャッシュに入っていない述語よりもずっと低コストで評価できます。特に、その述語に、スレッド固有変数や文字列比較などの比較的コストの高い操作が含まれている場合、この傾向は顕著になります。通常、述語のキャッシュをユーザーが意識

することはありませんが、ここからは、より良い述語を作成するための指針を得ることができます。次の表を参照してください。

キャッシュ可能	キャッシュ不能
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->*any member* 、 curlwpsinfo->*any member*、 curpsinfo->*any member*

以下は、推奨しない例です。

```
syscall::read:entry
{
    follow[pid, tid] = 1;
}

fbt:::
/follow[pid, tid]/
{}

syscall::read:return
/follow[pid, tid]/
{
    follow[pid, tid] = 0;
}
```

次のように、スレッド固有変数を使用する方法を推奨します。

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

fbt:::
/self->follow/
{}

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

キャッシュ可能な式だけを含む述語のみが、キャッシュ可能です。以下の述語は、すべてキャッシュ可能です。

```
/execname == "myprogram"/
/execname == $$1/
/pid == 12345/
/pid == $1/
/self->traceme == 1/
```

以下の述語では、大域変数が使用されています。そのため、これらはすべてキャッシュ不能です。

```
/execname == one_to_watch/
/traceme[execname]/
/pid == pid_i_care_about/
/self->traceme == my_global/
```

◆ ◆ ◆ 第 39 章

39 安定性

Sun では、しばしば開発者を対象に、新しいテクノロジのアーリーアクセス版や、ユーザー/カーネルソフトウェアの内部実装の詳細を確認するための各種監視ツールを提供しています。しかし、残念なことに、新しいテクノロジや内部実装の詳細は、よく変更されます。インタフェースや実装は、ソフトウェアの更新やパッチの発行を経て、発展し成熟するからです。Sun では、attributes(5) のマニュアルページに記載されているラベルを使って、アプリケーションやインタフェースの安定性のレベルを文書化しています。ユーザーは、この情報を基に、将来のリリースでどのような変更が加えられる可能性があるかを予想できます。

ある D プログラムからアクセス可能なさまざまな構成要素やサービスについて、どれか 1 つの安定性属性で説明できるものではありません。このため、DTrace と D コンパイラには、ユーザーが作成する D プログラムの安定性レベルを、動的に計算し記述する機能が用意されています。この章では、プログラムの安定性を決定する DTrace 機能について説明します。この機能は、安定した D プログラムを設計する上で役立ちます。DTrace の安定性機能を利用して、D プログラムの安定性属性を確認したり、プログラムのインタフェースに有害な依存関係が生じたとき、コンパイル時エラーを発行することもできます。

安定性レベル

DTrace は、組み込み変数、関数、プローブなどの構成要素に対して、2 種類の安定性属性を提供しています。1 つは「安定性レベル」、もう 1 つはアーキテクチャの「依存クラス」です。DTrace の安定性レベルは、インタフェースや DTrace 構成要素が将来のリリースやパッチで変更される可能性を表しています。この情報は、DTrace ベースのスクリプトやツールを開発するときのリスク評価に役立ちます。DTrace の依存クラスは、あるインタフェースが、すべての Solaris プラットフォームおよびプロセッサに共通のインタフェースであるのか、あるいは特定のアーキテクチャ専用 (たとえば SPARC 専用) のインタフェースであるのかを示します。インタフェースの情報を提供するこの 2 種類の属性は、相互に依存することがありません。

以下では、DTrace で使用する安定性の値を、安定性の低いものから順に一覧します。比較的安定性の高いインタフェースは、すべての D プログラム、すべての階層化アプリケーションで使用できます。Sun では、これらのインタフェースが、将来のマイナーリリースでも引き続き動作するように開発を進めています。「安定」インタフェースだけに依存するアプリケーションは、将来のマイナーリリースでも正しく機能し、暫定パッチを適用しても問題は発生しません。比較的安定性の低いインタフェースは、現在のシステムでの計測、プロトタイピング、チューニング、およびデバッグには対応していますが、将来のマイナーリリースで、互換性の問題が発生したり、落とされたり、別のインタフェースに置き換えられたりする可能性があります。そのため、これらを使用するときは、注意が必要です。

DTrace の安定性の値は、DTrace インタフェースの安定性だけでなく、監視対象のソフトウェア構成要素の安定性を把握する上でも役立ちます。したがって、監視対象のソフトウェアスタックをアップグレードしたり変更したりするときに、D プログラムや階層化ツールの変更が必要になる可能性についても、DTrace の安定性の値から判断できます。

内部 (Internal)
: DTrace 専用のインタフェースです。DTrace の実装の詳細を示します。「内部」インタフェースは、マイナーリリースやマイクロリリースで変更される可能性があります。

非公開 (Private)
: Sun 専用のインタフェースです。Sun のほかの製品のために開発されたインタフェースで、まだ顧客や ISV 向けに正式に文書化されていません。「非公開」インタフェースは、マイナーリリースやマイクロリリースで変更される可能性があります。

廃止 (Obsolete)
: 現在のリリースではサポートされているが、将来的に削除される予定の (特にマイナーリリースで削除される可能性が高い) インタフェースです。Sun では、インタフェースのサポートを打ち切る場合には、なるべく事前に通知するようにしています。「廃止」インタフェースを使用しようとすると、D コンパイラから警告メッセージが出力されることがあります。

外部 (External)
: このインタフェースは Sun 以外によって管理されています。Sun は、これらのインタフェースの更新版 (ただし以前のものと互換性がない可能性がある) を管理元から入手できる場合には、独自の裁量で、これをリリースの一部に追加して配布できます。異なるリリース間での「外部」インタフェースのソースまたはバイナリ互換性については、Sun は一切関与しません。これらのインタフェースに基づくアプリケーション (「外部」インタフェースが含まれているパッチも含む) は、将来のリリースでは動作しない可能性があります。

変更の可能性あり (Unstable)
: 新しい (頻繁に変更される) テクノロジや、システム動作の監視やデバッグに必要な実装アーティファクトへの、開発者向けアーリーアクセス用に提供されているインタフェースです。将来的により安定したソリューションが提供される予定です。「変更の可

能性あり」インタフェースのマイナーリリース間でのソースまたはバイナリ互換性については、Sun は一切関与しません。

発展中 (Evolving)　将来的に「標準」インタフェースまたは「安定」インタフェースになる可能性があるが、まだ過渡期にあるインタフェースです。Sun は、発展後も以前のリリースと互換性が保たれるよう、適切な努力をします。上位互換性のない変更が必要な場合は、マイナーリリースとメジャーリリースで行われます。こうした変更は、マイクロリリースではできるかぎり回避されます。こうした変更の必要が生じた場合、影響を受けるリリースのリリースノートにその旨が文書化されます。可能なかぎり、Sun は、バイナリ互換性の実現と D プログラム開発続行のために、移行支援ツールを提供します。

安定 (Stable)　Sun の管理下にある完成したインタフェースです。Sun は、これらのインタフェースに対する上位互換性のない変更をできるかぎり回避します。特に、マイナーリリースとマイクロリリースでの変更を回避します。「安定」インタフェースのサポートを打ち切る場合、Sun はその旨を通知し、安定性レベルを「廃止」に変更します。

標準 (Standard)　業界標準に準拠したインタフェースです。このインタフェースの文書には、準拠する標準の説明が記載されています。標準は、通常、標準開発団体によって管理されています。この標準の変更が認可された場合、それに沿って「標準」インタフェースが変更されることがあります。この安定性レベルは、業界の慣例により、正規標準なしで採用されたインタフェースにも当てはまります。指定されたバージョンの標準のみがサポート対象になります。以降のバージョンがサポートされるかどうかは保証されていません。「標準」インタフェースについて、上位互換性のない変更が標準開発団体によって認可され、この変更をサポートすることを Sun が決定した場合には、Sun は互換性と移行方法を通知します。

依存クラス

Solaris と DTrace では、さまざまなオペレーティングプラットフォームとプロセッサがサポートされています。このため、DTrace では、インタフェースを「依存クラス」に分類して、すべての Solaris プラットフォームとプロセッサに共通のインタフェースであるのか、特定のシステムアーキテクチャ専用のインタフェースであるのかを区別しています。依存クラスは、上で説明した安定性レベルとは別の観点からの分類です。たとえば、安定しているが SPARC マイクロプロセッサでしかサポートされない DTrace インタフェースもあれば、変更の可能性はあるがすべての Solaris システムに共通の DTrace インタフェースもあります。以下では、DTrace の依存クラスについて、共通性の低いものから順に説明します。もっとも共通性が低いクラス

は、特定のアーキテクチャに固有のクラスです。もっとも共通性が高いクラスは、すべてのアーキテクチャに共通のクラスです。

不明 (Unknown)
: アーキテクチャの依存関係が不明なインタフェースです。DTrace は、すべての構成要素 (オペレーティングシステム実装内で定義されたデータ型など) について、アーキテクチャの依存関係を認識できるとは限りません。通常、「不明」に分類されるのは、依存関係を計算できない、安定性の非常に低いインタフェースです。現在使用中のアーキテクチャ以外で DTrace を使用するときには、このインタフェースが使用できない可能性があります。

CPU
: 現在のシステムの CPU モデル固有のインタフェースです。現在の CPU モデルと実装名を表示するには、psrinfo(1M) ユーティリティーの -v オプションを使用します。CPU モデル依存インタフェースは、その他の CPU 実装では使用できない可能性があります。同じ命令セットアーキテクチャ (ISA) をエクスポートする CPU でも同様です。たとえば、UltraSPARC-III+ と UltraSPARC-II は、どちらも SPARC 命令セットをサポートします。しかし、UltraSPARC-III+ マイクロプロセッサ上の CPU 依存インタフェースは、UltraSPARC-II マイクロプロセッサでは使用できない可能性があります。

プラットフォーム (Platform)
: 現在のシステムのハードウェアプラットフォーム固有のインタフェースです。通常、プラットフォームは、システムコンポーネントとアーキテクチャの特性のセット (たとえばサポート対象の CPU モデルのセット) に、「SUNW,Ultra-Enterprise-10000」のようなシステム名を関連付けています。現在のプラットフォーム名を表示するには、uname(1) -i オプションを使用します。このインタフェースは、ほかのハードウェアプラットフォームでは使用できない可能性があります。

グループ (Group)
: 現在のシステムのハードウェアプラットフォームグループ固有のインタフェースです。通常、プラットフォームグループは、プラットフォームのセットと関連する特性を、「sun4u」のような 1 つの名前で関連付けています。現在のプラットフォームグループ名を表示するには、uname(1) -m オプションを使用します。このインタフェースは、現在のプラットフォームグループ内のその他のプラットフォーム上では使用可能ですが、このグループのメンバーでないハードウェアプラットフォーム上では使用できない可能性があります。

ISA
: このシステム上のマイクロプロセッサがサポートする命令セットアーキテクチャ (ISA) 固有のインタフェースです。ISA は、マイクロプロセッサ上で実行可能なソフトウェアの仕様 (アセンブリ言語命令、レジスタなどの詳細情報を含む) について説明したものです。システムがサポートするネイティブの命令セットを表示す

	るには、isainfo(1) ユーティリティーを使用します。このインタフェースは、同じ命令セットをエクスポートしないシステム上ではサポートされない可能性があります。たとえば、Solaris SPARC システム上の ISA 依存インタフェースは、Solaris x86 システム上ではサポートされない可能性があります。
共通	配下のハードウェアとは関係なく、すべての Solaris システムに共通のインタフェースです。「共通」インタフェースだけに依存する DTrace プログラムと階層化アプリケーションは、Solaris と DTrace のリビジョンが共通しているその他の Solaris システムに配備し、実行できます。大部分の DTrace インタフェースは「共通」インタフェースです。したがって、Solaris を使用する場合いつでも使用できます。

インタフェース属性

DTrace は、インタフェースについて説明するとき、2 つの安定性レベルと 1 つの依存クラスからなる三つ組の属性を使用します。慣習上、インタフェース属性は次の順序で記述します。各属性は、スラッシュで区切ります。

name-stability / *data-stability* / *dependency-class*

インタフェースの「名前の安定性 *(name-stability)*」は、D プログラム内または dtrace(1M) コマンド行に表示されるインタフェース名の安定性レベルです。たとえば、D 変数名 `execname` は、安定しています。 Sun は、この識別子が、「安定」インタフェースの規則 (上記参照) に従って、D プログラム内で引き続きサポートされることを保証します。

インタフェースの「データの安定性 *(data-stability)*」は、名前の安定性とは別のものです。この安定性レベルは、インタフェースや関連データセマンティクスで使用されるデータ書式を保持する Sun の責任について説明するものです。たとえば、D 変数 `pid` は、「安定」インタフェースです。 プロセス ID は、Solaris で使用される安定した概念です。Sun は、`pid` 変数が `pid_t` 型であり、「安定」インタフェースの規則に従って、プローブを起動したスレッドのプロセス ID に設定されるというセマンティクスを持つことを保証します。

インタフェースの「依存クラス *(dependency-class)*」は、名前やデータの安定性とは別のものです。依存クラスは、このインタフェースが現在のオペレーティングプラットフォームやマイクロプロセッサに固有であるかどうかを示します。

DTrace と D コンパイラは、プロバイダ、プローブ記述、D 変数、D 関数、型、プログラム文自体を含む、DTrace インタフェースの全構成要素の安定性属性を追跡します。この 3 つの値は、互いに異なることがあります。たとえば、D 変数 `curthread` は、「Stable/Private/Common」という属性を持っています。 これは、変数名が「安

定」していて、すべての Solaris オペレーティングプラットフォームに「共通」であるが、この変数は、Solaris カーネルの実装アーティファクトである「非公開」のデータ書式へのアクセスを提供するという意味です。ほとんどの D 変数はユーザー定義の変数なので、「Stable/Stable/Common」の属性を持っています。

安定性の計算と報告

D コンパイラは、D プログラム内の個々のプローブ記述とアクション文に対して、安定性の計算を行います。プログラムの安定性のレポートを表示するには、`dtrace -v` オプションを使用します。以下の例では、コマンド行に記述されたプログラムを使用します。

```
# 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
```

プローブを有効にせず、プログラムも実行しないで、プログラムの安定性を判定したい場合は、`dtrace -v` オプションと `-e` オプションを組み合わせて使用します。この場合、dtrace は、D プログラムを実行せず、コンパイルだけを行います。以下に、別の安定性レポートの例を示します。

```
# 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
#
```

新しいプログラムでは、D 変数 `curthread` を参照しています。この D 変数は、安定した名前と非公開のデータセマンティクスを持っています。非公開のデータセマン

ティクスを持っているということは、この変数について調べるとき、カーネルの非公開の実装の詳細にアクセスするということです。プログラムの安定性レポートには、現在、この状態が反映されています。プログラムレポート内の安定性属性は、インタフェース属性の三つ組の対応する値から最小の安定性レベルとクラスを選択することで計算されます。

プローブ記述の安定性属性は、プロバイダによって発行された属性に従って、指定されたすべてのプローブ記述フィールドの最小の安定性属性をとることによって計算されます。使用可能な DTrace プロバイダの属性については、各プロバイダの章を参照してください。DTrace プロバイダは、自身が公開しているすべてのプローブの 4 つの記述フィールドの 1 つ 1 つに対して、安定性属性の三つ組をエクスポートします。このため、プロバイダの名前のほうが、このプロバイダがエクスポートする個々のプローブよりも安定性が高い場合があります。たとえば、次のようなプローブ記述があるとします。

```
fbt:::
```

このプローブ記述は、すべてのカーネル関数の開始 (entry) と終了 (return) をトレースするように指示するもので、次のプローブ記述より高い安定性を備えています。

```
fbt:foo:bar:entry
```

このプローブ記述は、カーネルモジュール `foo()` 内の内部関数 `bar` を指定しています。簡便性のため、ほとんどのプロバイダは、公開する値 *module*:*function*:*name* 全部に、単一の属性セットを使用します。プロバイダは、`args[]` 配列の属性を指定します。これは、プローブ引数の安定性がプロバイダごとに異なるからです。

プローブ記述のプロバイダフィールドが未指定の場合、この記述には安定性属性 Unstable/Unstable/Common が割り当てられます。これは、この記述が将来の Solaris バージョンで使用されたとき、現在まだ存在しないプロバイダのプローブと照合されてしまう可能性があるからです。このため、Sun は、このプログラムの将来の安定性と動作について保証できません。D プログラム節を作成するときは、プロバイダを常に明示的に指定する必要があります。また、パターンマッチング文字 (第 4 章「D プログラムの構造」を参照) や `$15` のようなマクロ変数 (第 15 章「スクリプトの作成」を参照) を含むプローブ記述フィールドは、未指定として扱われます。これは、これらの記述パターンを展開すると、将来の DTrace や Solaris OS で、Sun によってリリースされたプロバイダまたはプローブと照合されてしまう可能性があるからです。

ほとんどの D 言語の文で、安定性属性は、その文の構成要素の最小の安定性とクラスをとることで計算されます。たとえば、次の D 言語構成要素は、次の属性を持っています。

構成要素	属性
D 組み込み変数 `curthread`	Stable/Private/Common
D ユーザー定義変数 `x`	安定/安定/共通

次の D プログラム文を作成するとします。

```
x += curthread->t_pri;
```

この文の属性は、「Stable/Private/Common」になります。これは、オペランド `curthread` と `x` の最小属性です。式の安定性は、各オペランドの最小属性をとることで計算されます。

プログラム内に定義した D 変数には、属性「Stable/Stable/Common」が自動的に割り当てられます。D 言語の文法や D 演算子には、「Stable/Stable/Common」の属性が暗黙的に割り当てられます。逆引用符演算子 (`) によるカーネルシンボルの参照には、常に「Private/Private/Unknown」の属性が割り当てられます。これは、これらが実装アーティファクトを反映しているからです。D プログラムソースコード内で定義した型、具体的に言うと、C 型や D 型の名前空間に関連付けられた型には、「Stable/Stable/Common」の属性が割り当てられます。オペレーティングシステム実装内に定義され、ほかの型の名前空間から提供された型には、「Private/Private/Unknown」の属性が割り当てられます。D 型キャスト演算子が生成する式には、入力式の安定性属性とキャスト出力型の安定性属性との最小値が割り当てられます。

C プリプロセッサを使って C システムヘッダーファイルをインクルードする場合、これらの型には、C 型名前空間が関連付けられ、属性「Stable/Stable/Common」が割り当てられます。これは、D コンパイラが、ユーザーがこれらの宣言に責任を持つと仮定せざるをえないからです。このため、C プリプロセッサを使って実装アーティファクトを含むヘッダーファイルをインクルードする場合、プログラムの安定性を誤る可能性があります。正しい安定性レベルを判断するには、インクルードするヘッダーファイルの文書を常に参照する必要があります。

安定性の強制

DTrace スクリプトや階層化ツールを開発するとき、安定性の問題の原因を特定したり、プログラムの安定性属性のセットが適切であることを確認したりできます。属性の計算結果として、コマンド行に指定した最小値未満の三つ組が得られた場合に、D コンパイラに強制的にエラーを返させるには、`dtrace -x amin=`*attributes* オプションを使用します。以下では、D プログラムソースの抜粋を使って、`-x amin` の使用例を紹介します。属性は、通常の順番で、スラッシュ (`/`) で区切った形式で指定されています。

```
# 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
#
```

◆ ◆ ◆ 第 40 章

40 トランスレータ

第 39 章「安定性」では、DTrace を使ってプログラムの安定性属性を計算し、報告する方法について学びました。本来なら、「安定」または「発展中」のインタフェースだけを使用して DTrace プログラムを作成するのが理想です。しかし、残念ながら、下位レベルの問題のデバッグ時やシステムパフォーマンスの測定時には、システムコールのような安定性の高いインタフェースに関連付けられているプローブではなく、カーネル内の関数のような内部オペレーティングシステムルーチンに関連付けられているプローブを有効にしなければならない場合があります。ソフトウェアスタック内の深いところにあるプローブのデータは、通常、実装アーティファクトの集合であって、Solaris システムコールインタフェースのデータのような、安定したデータ構造は備えていません。安定した D プログラムの作成を支援するため、DTrace では、実装アーティファクトを D プログラム文からアクセスできる安定したデータ構造に翻訳する機能を提供します。

トランスレータの宣言

「トランスレータ」は、インタフェースの提供元から提供される D 代入文の集合です。これを使って、入力式を構造型のオブジェクトに翻訳できます。以下では、`stdio.h` に定義された ANSI-C 標準ライブラリルーチンの例を使って、トランスレータの必要性および使用方法について説明します。これらのルーチンは、`FILE` という名前のデータ構造に作用します。FILE の実装アーティファクトは、C プログラマによって抽出されています。データ構造の抽象オブジェクトは、公開ヘッダーファイルでデータ構造の前方宣言だけを行い、対応する構造体定義を別の非公開ヘッダーファイルに保存することによって作成できます。これが標準の方法です。

C プログラムの作成中に、`FILE` 構造体のファイル記述子について調べる場合は、FILE 構造体のメンバーを間接参照するのではなく、`fileno(3C)` 関数を使って記述子を取得します。Solaris ヘッダーファイルは、この規則を徹底させるため、FILE を不透明な前方宣言タグとして定義しています。このため、`<stdio.h>` を含む C プログラムから FILE を間接参照することはできません。`libc.so.1` ライブラリ内の `fileno()` が、次のような C プログラムによって実装されていると仮定します。

```
int
fileno(FILE *fp)
{
    struct file_impl *ip = (struct file_impl *)fp;

    return (ip->fd);
}
```

この例の `fileno()` は、引数として `FILE` ポインタをとり、これを対応する内部 `libc` 構造のポインタ `struct file_impl` へキャストします。そして、実装構造の `fd` メンバーの値を返します。なぜ Solaris には、このようなインタフェースが実装されているのでしょうか。それは、Sun がクライアントプログラムから現在の `libc` 実装の詳細を抽出することにより、継続的に `libc` の内部実装の詳細を発展させ、変更しながら、強力なバイナリ互換性の実現に努めることができるからです。上の例で、`fd` メンバーのサイズや `struct file_impl` 内での位置は、パッチで変更されることすらありえます。しかし、`fileno(3C)` を呼び出す既存のバイナリは、これらのアーティファクトに依存していないため、このような変更の影響を受けません。

残念ながら、DTrace のような監視機能付きソフトウェアでは、実装内部に注目しないと有効な結果が得られません。このため、Solaris ライブラリ内やカーネル内に定義されている任意の C 関数を呼び出す余裕はありません。`stdio.h` に宣言されているルーチンを実装するために、D プログラム内に `struct file_impl` のコピーを宣言することも可能ですが、その場合 D プログラムはライブラリの非公開実装アーティファクトに依存することになります。こうした非公開実装アーティファクトは、将来のマイクロリリースやマイナーリリースで、場合によってはパッチを適用しただけで、使用できなくなる可能性があります。理想的な D プログラム用構文は、ライブラリの実装に結合され、適宜更新されながらも、より安定性の高い抽象層を提供できるような構文です。

新しいトランスレータを作成するときは、次のような宣言を使用します。

```
translator output-type < input-type input-identifier > {
    member-name = expression ;
    member-name = expression ;
    ...
};
```

output-type には、構造体を指定します。これが、翻訳の結果の型になります。*input-type* には、入力式の型を指定し、山括弧 (< >) で囲んで、トランスレータ式で入力式の別名として使用できる入力識別子 *input-identifier* を 1 つ追加します。トランスレータの本体は、中括弧 ({ }) 内に記述され、セミコロン (;) で終わります。トランスレータを構成するのは、メンバー名 (*member-name*) と、翻訳式に対応する識別子です。メンバー宣言では、出力型 *output-type* の固有のメンバーを指定する必要があります。また、D 代入演算子 (=) の規則に従って、指定したメンバーの型と互換性のある型の式を割り当てる必要があります。

たとえば、使用可能ないくつかの libc インタフェースに基づいて、stdio ファイルについての安定した情報の構造体を定義します。

```
struct file_info {
    int file_fd;   /* file descriptor from fileno(3C) */
    int file_eof;  /* eof flag from feof(3C) */
};
```

続いて、FILE を file_info へ翻訳する D トランスレータを宣言します。

```
translator struct file_info < FILE *F > {
    file_fd = ((struct file_impl *)F)->fd;
    file_eof = ((struct file_impl *)F)->eof;
};
```

このトランスレータでは、FILE * 型の入力式に、入力識別子 F が割り当てられます。その後、識別子 F をトランスレータメンバー式で FILE * 型の変数として使用できます。この変数は、トランスレータ宣言の本体内でしか、可視的に使用されません。出力 file_fd メンバーの値を特定するため、トランスレータは、上記の fileno(3C) の実装例と同じように、キャストと間接参照を行います。EOF 指示子の値を得るときも、同様の翻訳が行われます。

Sun では、D プログラムから起動可能な、Solaris インタフェース用のトランスレータセットを提供しています。これらのトランスレータは、対応するインタフェースの実装が変更されても、以前に定義されたインタフェースの安定性規則に従って保持されます。これらのトランスレータについては、D でトランスレータを呼び出す方法を説明したあと説明します。トランスレータ機能自体は、ソフトウェアパッケージの状態の監視用の独自のトランスレータを D プログラマに提供したいと考える、アプリケーション開発者やライブラリ開発者も、対象としています。

翻訳演算子

D 演算子 xlate は、入力式を定義済みの翻訳出力構造へ翻訳するときに使用します。xlate 演算子の使用形式は、次のとおりです。

```
xlate < output-type > ( input-expression )
```

たとえば、先ほど定義した FILE 構造体のトランスレータを呼び出し、file_fd メンバーにアクセスしたい場合は、次のような式を作成します。

```
xlate <struct file_info *>(f)->file_fd;
```

f は FILE * 型の D 変数です。xlate 式そのものには、*output-type* で定義された型が割り当てられます。トランスレータの定義後は、このトランスレータを使って、入力式を出力構造型またはこの構造型のポインタへ翻訳できます。

入力式を構造体に翻訳する場合は、演算子「.」を使って、特定の出力メンバーを間接参照するか、翻訳済みの構造体全体を別のD変数に割り当てて、すべてのメンバーの値のコピーを作成するかします。単一のメンバーを間接参照する場合、Dコンパイラは、そのメンバーの式に対応するコードしか生成しません。翻訳済みの構造体に&演算子を適用して、そのアドレスを取得することはできません。というのも、コピーが作成されるか、いずれかのメンバーが参照されるまで、データオブジェクト自体が存在しないからです。

入力式を構造体のポインタに翻訳する場合は、演算子「->」を使って、出力の特定メンバーを間接参照するか、単項演算子「*」を使ってポインタを間接参照するかします。後者の場合、式を構造体に翻訳した場合と同じ結果が得られます。単一のメンバーを間接参照する場合、Dコンパイラは、そのメンバーの式に対応するコードしか生成しません。翻訳済みのポインタを別のD変数に割り当てることはできません。というのも、コピーが作成されるか、いずれかのメンバーが参照されるまで、データオブジェクト自体が存在しないからです。存在しないデータオブジェクトにアドレスを指定することはできません。

トランスレータ宣言では、1つ以上の出力型のメンバーの式を省略できます。たとえば、`xlate`式を使って、翻訳式が定義されていないメンバーにアクセスしようとすると、Dコンパイラからエラーメッセージが返され、プログラムのコンパイルが中止されます。構造体の割り当てによって出力型全体をコピーした場合、翻訳式が定義されていないメンバーはゼロで初期化されます。

`xlate`操作に適したトランスレータを見つけ出すため、Dコンパイラは、使用可能なトランスレータを次の順番でチェックします。

- まず、コンパイラは、入力式の型から出力型への翻訳を探します。
- 次に、コンパイラは、配下の型名の型付きの別名に従って、入力型と出力型の「解決」を行います。その後、解決済み入力型から解決済み出力型への翻訳を探します。
- その次に、コンパイラは、互換性のある入力型から解決済みの出力型への翻訳を探します。コンパイラは、関数呼び出しの引数と関数のプロトタイプの互換性について判断するときと同じ規則に従って、入力式の型とトランスレータの入力型に互換性があるかどうかを判断します。

これらの規則に基づいて一致するトランスレータが見つからない場合は、Dコンパイラからエラーメッセージが出力され、プログラムのコンパイルが失敗します。

プロセスモデルトランスレータ

DTrace ライブラリファイル /usr/lib/dtrace/procfs.d は、D プログラム内で使用するトランスレータのセットを提供します。これらのトランスレータは、プロセスやスレッドのオペレーティングシステムカーネルの実装構造を、安定した proc(4) 構造である psinfo や lwpsinfo に翻訳します。これらの構造は、Solaris の /proc ファイルシステムファイル /proc/*pid*/psinfo や /proc/*pid*/lwps/*lwpid*/lwpsinfo でも使用されます。また、これらの定義は、システムヘッダーファイル /usr/include/sys/procfs.h に格納されています。これらの構造は、プロセスやスレッドについての安定した役立つ情報を定義します。たとえば、プロセス ID、LWP ID、初期引数、および ps(1) コマンドで出力されるその他のデータがこれに該当します。構造体のメンバーと意味は、proc(4) のマニュアルページで確認できます。

表 40–1 procfs.d トランスレータ

入力型	入力型の属性	出力型	出力型の属性
proc_t *	非公開/非公開/共通	psinfo_t *	安定/安定/共通
kthread_t *	非公開/非公開/共通	lwpsinfo_t *	安定/安定/共通

安定した翻訳

トランスレータは、情報を安定したデータ構造に変換する機能を備えています。しかし、データの翻訳時に発生するあらゆる安定性の問題を解決できるわけではありません。たとえば、xlate 操作の入力式そのものが、変更の可能性があるデータを参照している場合、結果の D プログラムも変更される可能性があります。これは、プログラムの安定性は、常に、蓄積された D プログラム文と式の最小限の安定性として計算されるためです。そのため、安定したプログラムを作成するためには、トランスレータに安定した入力式を定義する必要があります。D インライン機構は、「安定した翻訳」を促進するために使用されます。

procfs.d ライブラリは、以前に安定した翻訳として紹介した curlwpsinfo 変数と curpsinfo 変数を提供します。たとえば、curlwpsinfo 変数は、実際には次のように inline 宣言されます。

```
inline lwpsinfo_t *curlwpsinfo = xlate <lwpsinfo_t *> (curthread);
#pragma D attributes Stable/Stable/Common curlwpsinfo
```

curlwpsinfo 変数は、curthread 変数、スレッドを表すカーネルの非公開データ構造のポインタ、安定した lwpsinfo_t 型により、インライン翻訳として定義されています。D コンパイラは、このライブラリファイルを処理し、inline 宣言をキャッシュに格納します。これにより、curlwpsinfo の見た目はその他の D 変数と同じになります。この宣言の後ろの #pragma 文は、curlwpsinfo 識別子の属性を明示的に「Stable/Stable/Common」にリセットし、インライン式に含まれる curthread の参

照をマスクします。この D 機能の組み合わせにより、D プログラマは、Solaris 実装に変更が加えられるたびに更新される可能性のある curthread を、翻訳ソースとして安全に使用することができます。

◆ ◆ ◆ 第 41 章

バージョン管理

第 39 章「安定性」では、DTrace 機能を使って、自作の D プログラムの安定性属性を確認する方法を学びました。適切な安定性属性を持つ D プログラムを作成したあと、このプログラムを特定の「バージョン」の D プログラミングインタフェースに結合できます。D インタフェースのバージョンとは、D コンパイラから提供される型、変数、関数、定数、およびトランスレータから成る特定のセットに適用される、ラベルのようなものです。特定のバージョンの D プログラミングインタフェースへの結合を指定すると、次回以降のバージョンの DTrace でこのプログラムを再コンパイルする際、新しいバージョンの D プログラミングインタフェースで定義されるプログラム識別子と、今回ユーザーが定義したプログラム識別子の衝突が起こらなくなります。持続的なスクリプト (第 15 章「スクリプトの作成」を参照) としてインストールする D プログラムや、階層化ツールで使用したい D プログラムには、バージョンの結合を指定します。

バージョンとリリース

D コンパイラは、「バージョン文字列」を使って、特定のソフトウェアリリースに対応する型、変数、関数、定数、およびトランスレータから成るセットにラベルを付けます。バージョン文字列は、10 進整数をピリオドで区切った形式の文字列です。メジャーリリースは「*x*」、マイナーリリースは「*x.y*」、マイクロリリースは「*x.y.z*」の形式をとります。バージョンを比較するときは、バージョン文字列の整数を左から順に比較します。一番左の整数の値が大きいほうが新しいバージョンです。一番左の整数の値が同じである場合は 1 つ右の整数、この整数の値も同じである場合はもう 1 つ右の整数が比較されます。バージョン文字列の一部が指定されていない場合、その部分にはゼロが入っているものと解釈して比較されます。

DTrace のバージョン文字列は、Sun のインタフェースのバージョンの標準的な命名法に従って決定されます。`attributes(5)` を参照してください。D プログラミングインタフェースに変更が加えられると、バージョン文字列が更新されます。以下の表に、DTrace のバージョン文字列と、このバージョン文字列で表される DTrace ソフトウェアリリースの意味をまとめます。

表 41–1 DTrace リリースバージョン

リリース	バージョン	意味
メジャー	*x*.0	メジャーリリースでは、通常、主要機能が追加され、さまざまな「標準」の改訂 (互換性のない場合もある) への対応が盛り込まれます。まれに、「標準」インタフェースや「安定」インタフェースが変更されたり落とされたり置換されたりする場合もあります (第 39 章「安定性」を参照)。D プログラミングインタフェースの初期バージョンは「1.0」です。
マイナー	*x.y*	*x*.0 や以前のバージョンと比較して、新しいマイナーリリース (y はゼロ以外) に通常含まれるのは、主要でない機能の追加のほか、「標準」インタフェースや「安定」インタフェースの互換性のある変更、「発展中」インタフェースの互換性のない場合もある変更、「変更の可能性あり」インタフェースのおそらく互換性のない変更などです。これらの変更には、新しい組み込み D 型、変数、関数、定数、およびトランスレータが含まれることがあります。また、マイナーリリースでは、以前に「廃止」というラベルが付けられたインタフェースがサポートされなくなることがあります (第 39 章「安定性」を参照)。
マイクロ	*x.y.z*	マイクロリリース (z はゼロ以外) では、以前のリリースとインタフェースの互換性が保たれる一方で、通常、バグ修正、パフォーマンス拡張機能、ハードウェアの追加サポートなどが含まれます。

通常、D プログラミングインタフェースの新しいバージョンでは、以前のバージョンで提供されていた機能の上位集合が提供されます。ただし、削除された「廃止」インタフェースは例外です。

バージョン管理オプション

デフォルトでは、`dtrace -s` または `dtrace -P`、`-m`、`-f`、`-n`、`-i` などを指定してコンパイルした D プログラムは、D コンパイラが提供する最新の D プログラミングインタフェースバージョンに結合されます。現在の D プログラミングインタフェースのバージョンを確認するには、`dtrace -V` コマンドを実行します。

```
$ dtrace -V
dtrace: Sun D 1.0
$
```

特定のバージョンの D プログラミングインタフェースに結合したい場合は、適切なバージョン文字列に `version` オプションを設定します。その他の DTrace オプションと同じく (第 16 章「オプションとチューニング可能パラメータ」を参照)、バージョンオプションを設定するときは、コマンド行で `dtrace -x` を実行します。

```
# dtrace -x version=1.0 -n 'BEGIN{trace("hello");}'
```

`#pragma D` オプション構文を使って、D プログラムソースファイルにオプションを設定することもできます。

```
#pragma D option version=1.0

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

`#pragma D option` 構文を使ってバージョン結合を要求する場合は、この指令を D プログラムファイルの最上部、その他の宣言やプローブ節よりも前に置く必要があります。バージョン結合引数が有効なバージョン文字列でない場合、または D コンパイラから提供されていないバージョンを参照している場合、エラーメッセージが出力され、コンパイルは失敗します。このため、バージョン結合機能を使えば、古いバージョンの DTrace 上で D スクリプトが実行された場合に、わかりやすいエラーメッセージを出して失敗させることもできます。

D コンパイラは、プログラムの宣言や節をコンパイルする前に、インタフェースのバージョンに合った D 型、関数、定数、およびトランスレータから成るセットを、コンパイラ名前空間にロードします。したがって、プログラム内で定義されている変数、型、およびトランスレータに加えて、このプログラムからアクセスできる識別子、型、およびトランスレータから成るセットも、ユーザーが指定したバージョン結合オプションによって制御されます。バージョン結合されていると、プログラムのソースコード内の宣言と矛盾する (したがってコンパイルエラーの原因となる) 識別子やトランスレータの定義が含まれるような新しいインタフェースが、D コンパイラによってロードされることはありません。将来のバージョンの DTrace が提供するインタフェースと矛盾しない識別子名を選択する方法については、49 ページの「識別子名とキーワード」を参照してください。

プロバイダのバージョン管理

DTrace プロバイダが提供するインタフェース (プローブとプローブ引数) は、D コンパイラが提供するインタフェースとは異なり、D プログラミングインタフェースや、先ほど説明したバージョン結合オプションの影響を受けません。使用可能なプロバイダインタフェースは、オペレーティングシステムカーネル内の DTrace ソフトウェアにコンパイル済みの計測機能をロードするときに決定されます。これらは、使用している命令セットアーキテクチャ、オペレーティングプラットフォーム、プロセッサ、Solaris システムにインストールされているソフトウェア、および現在のセキュリティ権限によって異なります。D コンパイラと DTrace ランタイムは、D プログラム節内のプローブを検査し、D プログラムから要求されたプローブが使用できない場合はエラーメッセージを返します。これらの機能は、D プログラミングインタフェースのバージョンとは異なる概念に基づいています。これは、DTrace プロバイダが、D プログラム内の定義と矛盾するインタフェースをエクスポートしないからです。言い換えると、D では、プローブを有効化することはできても定義することはできず、プローブ名は、その他の D プログラム識別子とは別の名前空間に格納されます。

配布される DTrace プロバイダは、Solaris リリースによって異なります。該当するリリース用の『Solaris 動的トレースガイド』を参照してください。このマニュアルの各プロバイダごとの章には、変更点や新機能の説明も含まれます。Solaris システム上で使用できるプロバイダとプローブについて調べるには、`dtrace -l` を使用します。プロバイダは、DTrace の安定性属性を使って、インタフェースにラベルを付けます。ユーザーは、DTrace の安定性報告機能 (第 39 章「安定性」を参照) を使って、D プログラムで使用されているプロバイダインタフェースが将来の Solaris リリースでも引き続き提供されるか、あるいは変更されるかを判断できます。

用語集

DTrace 任意の問題に対して簡潔な答えを提示する動的トレース機能。

アクション DTrace フレームワークによって実行される動作。この動作はプローブ起動時に実行され、その結果、データのトレースや、DTrace 外部のシステム状態の変更が行われます。アクションとして分類される動作には、データのトレース、プロセスの停止、スタックトレースの捕捉などがあります。

コンシューマ DTrace を使って計測機能を有効化し、トレースデータから結果ストリームを読み取るプログラム。`dtrace` コマンドは、標準の DTrace コンシューマです。`lockstat(1M)` ユーティリティーは、特殊な DTrace コンシューマです。

サブルーチン DTrace フレームワークによって実行される動作。この動作はプローブ起動時に実行され、DTrace 内部の状態を変更します。このとき、データのトレースは行われません。サブルーチンの要求には、「アクション」の場合と同様に、D 関数呼び出し構文が使用されます。

集積体 第 9 章「集積体」で定義した「集積関数」の結果を格納するオブジェクト。複数の式から成る組でインデックスが付けられ、結果をまとめるときに使用します。

述語 プローブの起動時に、一連のトレースアクションを実行するかどうかを決定付ける論理式。各 D プログラム節には、スラッシュ `/ /` で囲まれた述語が 1 つずつ割り当てられます。

節 プローブ指定子のリスト、述語 (オプション)、アクション文のリスト (オプション) から成る、D プログラムの宣言。中括弧 (`{ }`) で囲んだ形式で表します。

トランスレータ 特定の計測されるサブシステムの実装の詳細を、入力式より安定性の高いインタフェースを形成する `struct` 型のオブジェクトに変換する D 代入文の集合。

プローブ DTrace が述語とアクションを含む計測機能を動的に結合する、システム内の場所またはアクティビティ。各プローブは、そのプロバイダ、モジュール、関数、および意味上の名前を指定する組で表されます。特定のモジュールや関数へ「アンカーされた」プローブと、特定のプログラムの場所に関連しない「アンカーされていない」プローブ (`profile` タイマーなど) とがあります。

プロバイダ DTrace フレームワークに代わって特定の種類の計測機能を実装するカーネルモジュール。プロバイダは、プローブの名前空間と、その名前およびデータのセマンティクスを表す安定性マトリックスをエクスポートします。詳細は、本マニュアルの該当する章を参照してください。

有効化	有効化された複数のプローブと、これらに関連付けられた述語およびアクションのグループ。

索引

E

F

I

K

L

は

ひ

ふ

ゆ

よ

れ