



# DTrace ユーザーガイド



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

Part No: 819-6259-10  
2006年5月

Sun Microsystems, Inc. (以下米国 Sun Microsystems 社とします) は、本書に記述されている製品に含まれる技術に関連する知的財産権を所有します。特に、この知的財産権はひとつかそれ以上の米国における特許、あるいは米国およびその他の国において申請中の特許を含んでいることがあります。それらに限定されるものではありません。

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company, Ltd. が独占的にライセンスしている米国ならびに他の国における登録商標です。フォント技術を含む第三者のソフトウェアは、著作権により保護されており、提供者からライセンスを受けているものです。

U.S. Government Rights Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

この配布には、第三者によって開発された素材を含んでいることがあります。

本製品に含まれる HG-MinchoL、HG-MinchoL-Sun、HG-PMinchoL-Sun、HG-GothicB、HG-GothicB-Sun、および HG-PGothicB-Sun は、株式会社リコーがリコービマジクス株式会社からライセンス供与されたタイプフェースマスタをもとに作成されたものです。HeiseiMin-W3H は、株式会社リコーが財団法人日本規格協会からライセンス供与されたタイプフェースマスタをもとに作成されたものです。フォントとして無断複製することは禁止されています。

Sun、Sun Microsystems、Sun のロゴマーク、Solaris のロゴマーク、Java Coffee Cup のロゴマーク、docs.sun.com、Java および Solaris は、米国およびその他の国における米国 Sun Microsystems 社の商標、登録商標もしくは、サービスマークです。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

OPENLOOK、OpenBoot、JLE は、サン・マイクロシステムズ株式会社の登録商標です。

Wnn は、京都大学、株式会社アステック、オムロン株式会社で共同開発されたソフトウェアです。

Wnn6 は、オムロン株式会社、オムロンソフトウェア株式会社で共同開発されたソフトウェアです。Copyright(C) OMRON Co., Ltd. 1995-2000. All Rights Reserved. Copyright(C) OMRON SOFTWARE Co., Ltd. 1995-2002 All Rights Reserved.

「ATOK for Solaris」は、株式会社ジャストシステムの著作物であり、「ATOK for Solaris」にかかる著作権、その他の権利は株式会社ジャストシステムおよび各権利者に帰属します。

「ATOK」および「推測変換」は、株式会社ジャストシステムの登録商標です。

「ATOK for Solaris」に添付するフェイスマーク辞書は、株式会社ビレッジセンターの許諾のもと、同社が発行する『インターネット・パソコン通信フェイスマークガイド』に添付のものを使用しています。

「ATOK for Solaris」に含まれる郵便番号辞書(7桁/5桁)は日本郵政公社が公開したデータを元に制作された物です(一部データの加工を行なっています)。

Unicode は、Unicode, Inc. の商標です。

本書で参照されている製品やサービスに関しては、該当する会社または組織に直接お問い合わせください。

OPEN LOOK および Sun Graphical User Interface は、米国 Sun Microsystems 社が自社のユーザおよびライセンス実施権者向けに開発しました。米国 Sun Microsystems 社は、コンピュータ産業用のビジュアルまたはグラフィカル・ユーザインタフェースの概念の研究開発における米国 Xerox 社の先駆者としての成果を認めるものです。米国 Sun Microsystems 社は米国 Xerox 社から Xerox Graphical User Interface の非独占的ライセンスを取得しており、このライセンスは、OPEN LOOK のグラフィカル・ユーザインタフェースを実装するか、またはその他の方法で米国 Sun Microsystems 社との書面によるライセンス契約を遵守する、米国 Sun Microsystems 社のライセンス実施権者にも適用されます。

本書で言及されている製品や含まれている情報は、米国輸出規制法で規制されるものであり、その他の国の輸出入に関する法律の対象となることがあります。核、ミサイル、化学あるいは生物兵器、原子力の海洋輸送手段への使用は、直接および間接を問わず厳しく禁止されています。米国が禁輸の対象としている国や、限定はされませんが、取引禁止顧客や特別指定国民のリストを含む米国輸出排除リストで指定されているものへの輸出および再輸出は厳しく禁止されています。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含みそれに限定されない、明示的であるか黙示的であるかを問わない、なんらの保証も行われぬものとします。

本製品が、外国為替および外国貿易管理法(外為法)に定められる戦略物資等(貨物または役務)に該当する場合、本製品を輸出または日本国外へ持ち出す際には、サン・マイクロシステムズ株式会社の事前の書面による承諾を得ることのほか、外為法および関連法規に基づく輸出手続き、また場合によっては、米国商務省または米国所轄官庁の許可を得ることが必要です。

原典: DTrace User Guide

Part No: 819-5488-10

Revision A

# 目次

---

はじめに .....	5
<b>1 概要 .....</b>	<b>9</b>
DTrace の機能 .....	9
アーキテクチャの概要 .....	10
DTrace プロバイダ .....	10
DTrace プローブ .....	11
DTrace の述語 .....	11
DTrace アクション .....	11
D スクリプト言語 .....	11
<b>2 DTrace の基本 .....</b>	<b>13</b>
プローブの一覧を表示する .....	13
DTrace でプローブを指定する .....	16
プローブを有効にする .....	16
DTrace アクションの基本 .....	18
データ記録アクション .....	20
破壊アクション .....	22
DTrace 集積体 .....	24
DTrace 集積体の構文 .....	25
<b>3 D 言語を使ったスクリプトの作成 .....</b>	<b>27</b>
D スクリプトの作成 .....	27
実行可能な D スクリプト .....	27
D リテラル文字列 .....	28
引数を使用する D スクリプトを作成する .....	30
DTrace の組み込み変数 .....	33

---

<b>4 DTrace の使用法</b> .....	37
パフォーマンス監視 .....	37
sysinfo プロバイダを使ってパフォーマンスの問題を検査する .....	37
ユーザープロセスをトレースする .....	44
サブルーチン copyin() と copyinstr() を使用する .....	44
dtrace の干渉を排除する .....	45
syscall プロバイダ .....	45
ustack() アクション .....	47
pid プロバイダ .....	48
匿名トレース .....	52
匿名有効化 .....	52
匿名状態を要求する .....	52
匿名トレースの例 .....	53
投機トレース .....	56
投機インタフェース .....	56
投機の作成 .....	57
投機の使用 .....	57
投機のコミット .....	58
投機の破棄 .....	58
投機の例 .....	58
 索引 .....	 65

# はじめに

---

『DTrace ユーザーガイド』は、トレースと分析のための強力なツール、DTrace についてわかりやすく説明する入門書です。このマニュアルでは、Dtrace の概要とその機能について説明します。さらに、DTrace を使って、比較的単純で一般的なタスクを実行する方法についても説明します。

## 対象読者

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 機能を利用して、ユーザープログラムの動作やオペレーティングシステムの動作を検査できます。DTrace を使用できるのは、実行中の本稼働システムのシステム管理者またはアプリケーション開発者です。

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

このマニュアルは、DTrace や D スクリプト言語に関するすべての情報を網羅するものではありません。より詳しい情報については、『Solaris 動的トレースガイド』を参照してください。

## お読みになる前に

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、[6 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。

## 関連情報

DTraceの詳細は、『Solaris 動的トレースガイド』を参照してください。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
- 『SOLARIS インターナルカーネル構造のすべて』、Jim Mauro、Richard McDougall 共著、ピアソン・エデュケーション発行、2001年、ISBN 4-89471-458-2
- 『最前線 UNIX のカーネル』、Uresh Vahalia 著、ピアソン・エデュケーション発行、2000年、ISBN 4-89471-189-3

## 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せん。

表 P-1 表記上の規則

字体または記号	意味	例
<b>AaBbCc123</b>	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。	<code>.login</code> ファイルを編集します。 <code>ls -a</code> を使用してすべてのファイルを表示します。  <code>system%</code>
<b>AaBbCc123</b>	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。	<code>system% su</code> <code>password:</code>
<i>AaBbCc123</i>	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、 <code>rm filename</code> 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。

表 P-1 表記上の規則 (続き)

字体または記号	意味	例
「」	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。	<code>sun% grep '^#define \  XV_VERSION_STRING'</code>

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。





# 概要

---

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 を使用できるのは、システム管理者またはアプリケーション開発者です。実行中の本稼働システム上で安全に使用できます。DTrace には、ユーザープログラムの動作やオペレーティングシステムの動作を検査する機能があります。DTrace ユーザーは、D スクリプト言語を使ってカスタムプログラムを作成できます。カスタムプログラムを使って、システムを動的に計測できます。カスタムプログラムを使用すると、特定のアプリケーションの動作に関する問題に対して、簡潔な答えをすばやく得ることができます。

## DTrace の機能

DTrace フレームワークでは、「プローブ」と呼ばれる計測ポイントを使用します。DTrace ユーザーは、プローブを使って、カーネルプロセスやユーザープロセスの関連情報を記録したり表示したりできます。それぞれの DTrace プローブは、特定の動作によって有効になります。プローブを有効にすることを「起動」と呼びます。たとえば、任意のカーネル関数に値が渡されたときに起動するプローブがあるとします。このサンプルプローブでは、次の情報を表示できます。

- 関数に渡されたすべての引数
- カーネル内のすべての大域変数
- 関数が呼び出された日時を示すタイムスタンプ
- 関数を呼び出したコードセクションを示すスタックトレース
- 関数が呼び出されたとき実行中だったプロセス
- 関数を呼び出したスレッド

プローブの起動時に DTrace によって実行される「アクション」を指定できます。DTrace のアクションは、通常、タイムスタンプや関数の引数といった重要なシステム動作を記録します。

プローブは、「プロバイダ」によって実装されます。プローブのプロバイダは、特定のプローブを起動可能にするカーネルモジュールです。たとえば、関数境界トレースプロバイダ `fbt` からは、全カーネルモジュール内のほぼすべての関数の開始 (entry) プローブと終了 (return) プローブが提供されています。

DTraceには、多数のデータ管理機能が用意されています。DTraceユーザーは、これらの機能を使ってプローブから報告されるデータを限定し、不要なデータの生成とフィルタリングにかかるオーバーヘッドをなくすことができます。DTraceには、ブート処理中にトレースを行ったり、カーネルのクラッシュダンプからデータを取得したりする機構もあります。DTraceの計測機能はすべて動的です。そのときに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。

DTraceフレームワークと対話するすべてのプロセスを、DTraceの「コンシューマ」と呼びます。Dtraceの一次コンシューマはdtrace(1M)ですが、これ以外のコンシューマも存在します。それらの追加コンシューマのほとんどは、lockstat(1M)のような既存のユーティリティの新規バージョンです。DTraceフレームワークでは、同時に実行できるコンシューマの数に制限はありません。

DTraceの動作は、D言語で記述されたスクリプトを使って変更できます。D言語は、C言語とよく似た構造を持つ言語です。D言語では、カーネルCデータ型、カーネル静的変数、およびカーネル大域変数にアクセスできます。D言語はANSIC演算子をサポートします。

## アーキテクチャーの概要

DTraceの機能は、次のコンポーネントで構成されています。

- dtraceのようなユーザーレベルのコンシューマプログラム
- トレースデータを収集するためのプローブを提供する、カーネルモジュールとしてパッケージ化されたプロバイダ
- コンシューマプログラムがdtrace(7D)カーネルドライバを使ってDTrace機能にアクセスするとき使用するライブラリインタフェース

## DTrace プロバイダ

プロバイダは、システムを計測する方法を示すものです。プロバイダは、DTraceフレームワークでプローブを使用できるようにします。DTraceはプロバイダに、プローブをいつ有効にするかという情報を送信します。プローブが有効にされて起動すると、プロバイダからDTraceに制御が移ります。

プロバイダは、カーネルモジュールのセットとしてパッケージ化されています。各モジュールは、特定の種類の計測機能を実行することにより、プローブを生成します。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できます。いずれかのプローブを発行したら、トレースアクションを有効にして、プローブに結合できます。

一部のプロバイダは、ユーザーからのトレース要求に基づいて、プローブを新規作成できます。

## DTrace プローブ

プローブには、次の属性があります。

- 「プロバイダ」によって提供される
- 計測対象の「モジュール」と「関数」を識別する
- 「名前」がある

この4つの属性のセットが、各プローブを一意に識別するプローブ指定子となります。次の形式で指定します。 `provider:module:function:name`。各プローブには、一意の整数値 ID も割り当てられています。

## DTrace の述語

述語とは、スラッシュ (/) で囲まれた式です。プローブの起動時に述語の評価が行われ、関連アクションを実行するべきかどうかが決まります。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。すべてのプローブで、プローブ節の述語部分は省略可能です。述語部分を省略した場合、そのプローブの起動のたびに指定のアクションが実行されます。

述語式では、任意の D 演算子を使用できます。述語式は、変数、定数などの D データオブジェクトを参照します。述語式の評価結果は、整数型またはポインタ型の値に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。

## DTrace アクション

アクションは、カーネル内で DTrace 仮想マシンが実行する文であり、ユーザーによるプログラムが可能です。アクションには、次のような性質があります。

- アクションはプローブの起動時に実行される
- アクションは D スクリプト言語で完全にプログラムできる
- ほとんどのアクションは、特定のシステム状態を記録する
- アクションを使って、システムの状態を記述内容どおりに変更することができる。このようなアクションは、「破壊アクション」と呼ばれている。デフォルトでは、破壊アクションは許可されない。
- 多くのアクションは、D スクリプト言語で記述された式を使用する

## D スクリプト言語

単純な関数の場合、コマンド行から直接 `dttrace` コマンドを使用して、DTrace フレームワークを呼び出すことができます。DTrace を使ってより複雑な関数を実行する場合は、

D スクリプト言語でスクリプトを作成します。DTrace で使用する特定のスクリプトをロードする場合は、`-s` オプションを使用します。D スクリプト言語の使用方法については、[第 3 章](#)を参照してください。

## DTrace の基本

---

この章では、いくつかの基本タスクの例を示しながら、DTrace の機能全般を紹介し  
ます。

### プローブの一覧を表示する

`dtrace` コマンドに `-l` オプションを指定すると、DTrace の全プローブの一覧を表示できま  
す。

```
# dtrace -l
ID PROVIDER MODULE FUNCTION NAME
1 dtrace BEGIN
2 dtrace END
3 dtrace ERROR
4 syscall nosys entry
5 syscall nosys return
6 syscall rexit entry
7 syscall rexit return
8 syscall forkall entry
9 syscall forkall return
10 syscall read entry
11 syscall read return
...
```

システム上で使用可能な全プローブの数を確認するには、次のコマンドを入力します。

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

報告されるプローブ数は、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。先ほどの例の `BEGIN` プローブと `END`  
プローブのように、`MODULE` 欄と `FUNCTION` 欄に何も表示されないことがあります。これら  
のフィールドが空になっているプローブは、特定の計測機能付きプログラムの関数や場  
所に対応していません。このようなプローブは、トレース要求の終わりなど、より抽象

的な概念を表しています。プローブ名にモジュールや関数が含まれている場合、このプローブを「アンカーされたプローブ」と呼びます。特定のモジュールや関数に関連付けられていないプローブを「アンカーされていないプローブ」と呼びます。

次の例のように、オプションを追加して、特定のプローブだけを一覧表示できます。

例2-1 関数を指定してプローブを一覧表示する

特定の関数に関連するプローブだけを一覧するには、`-f` オプションとその関数名を DTrace に指定します。

```
# dtrace -l -f cv_wait
ID      PROVIDER  MODULE      FUNCTION NAME
12921   fbt        genunix     cv_wait entry
12922   fbt        genunix     cv_wait return
```

例2-2 モジュールを指定してプローブを一覧表示する

特定のモジュールに関連するプローブだけを一覧するには、`-m` オプションとそのモジュール名を DTrace に指定します。

```
# dtrace -l -m sd
ID      PROVIDER  MODULE      FUNCTION NAME
17147   fbt        sd          sdopen entry
17148   fbt        sd          sdopen return
17149   fbt        sd          sdclose entry
17150   fbt        sd          sdclose return
17151   fbt        sd          sdstrategy entry
17152   fbt        sd          sdstrategy return
...
```

例2-3 名前を指定してプローブを一覧表示する

指定の名前のプローブを一覧するには、`-n` オプションとプローブ名を DTrace に指定します。

```
# dtrace -l -n BEGIN
ID      PROVIDER  MODULE      FUNCTION NAME
1       dtrace                                         BEGIN
```

例2-4 提供元のプロバイダを指定してプローブを一覧表示する

特定のプロバイダから提供されるプローブだけを一覧するには、`-P` オプションとそのプロバイダ名を DTrace に指定します。

例 2-4 提供元のプロバイダを指定してプローブを一覧表示する (続き)

```
# dtrace -l -P lockstat
ID      PROVIDER    MODULE      FUNCTION NAME
469     lockstat     genunix     mutex_enter adaptive-acquire
470     lockstat     genunix     mutex_enter adaptive-block
471     lockstat     genunix     mutex_enter adaptive-spin
472     lockstat     genunix     mutex_exit adaptive-release
473     lockstat     genunix     mutex_destroy adaptive-release
474     lockstat     genunix     mutex_tryenter adaptive-acquire
...
```

例 2-5 複数のプロバイダでサポートされる関数とモジュール

一部の関数またはモジュールは、次の例のように、複数のプロバイダでサポートされることがあります。

```
# dtrace -l -f read
ID      PROVIDER    MODULE      FUNCTION NAME
  10     syscall
  11     syscall
4036    sysinfo     genunix     read readch
4040    sysinfo     genunix     read sysread
7885     fbt         genunix     read entry
7886     fbt         genunix     read return
```

これまでの例が示すように、プローブ一覧の出力内容は次のとおりです。

- プローブに一意に割り当てられたプローブ ID (整数値)

---

注- プローブ ID の一意性は、Solaris オペレーティングシステムの同じリリース内または同じパッチレベル内でのみ確保されます。

---

- プロバイダ名
- モジュール名 (該当する場合)
- 関数名 (該当する場合)
- プローブ名

## DTraceでプローブを指定する

プローブを一意に識別する4つの要素を指定することにより、プローブを完全指定できます。プローブは次の形式で指定します。*provider:module:function:name*。プローブの要素の指定を省略すると、任意の要素を指定したことになります。たとえば、`fbt::alloc:entry` は、次の属性のプローブを指定しています。

- `fbt` プロバイダから提供されたプローブでなければならない
- 任意のモジュールに含まれるプローブでかまわない
- `alloc` 関数に含まれるプローブでなければならない
- プローブ名は `entry` でなければならない

4つの要素のうち、前半は省略可能です。`::open:entry` と指定した場合、`open:entry` と指定したのと同じことになります。どちらも、すべてのプロバイダおよびカーネルモジュールからの、関数名が `open`、プローブ名が `entry` であるプローブを指定したことになります。

```
# dtrace -l -n open:entry
  ID      PROVIDER      MODULE      FUNCTION NAME
  14      syscall           open entry
  7386    fbt                genunix     open entry
```

`sh(1)` のマニュアルページの「ファイル名の生成」のセクションに記載されている構文と同様のパターンマッチング構文を使って、プローブを指定することもできます。この構文では、特殊文字 `*`、`?`、`[`、`]`、および `]` を使用できます。プローブ記述 `syscall::open*:entry` は、`open` と `open64` の両方のシステムコールを意味します。`?` は、名前に含まれる任意の1文字を表します。`[と]` 文字は、名前に含まれる特定の文字列を指定するために使用します。

## プローブを有効にする

プローブを有効にするには、`-l` オプションを付けずにプローブを指定して、`dtrace` コマンドを実行します。指定のプローブが起動すると、DTrace はデフォルトのアクションを実行します。特別な指示は不要です。デフォルトのプローブアクションでは、指定のプローブが起動したことが示されます。それ以外のデータは記録されません。次に示すのは、`sd` モジュールに含まれるすべてのプローブを有効にするコード例です。

例 2-6 モジュールを指定してプローブを有効にする

```
# dtrace -m sd
CPU      ID      FUNCTION:NAME
  0      17329    sd_media_watch_cb:entry
  0      17330    sd_media_watch_cb:return
  0      17167    sdinfo:entry
  0      17168    sdinfo:return
  0      17151    sdstrategy:entry
```



例 2-6 モジュールを指定してプローブを有効にする (続き)

```

0 17152          sdstrategy:return
0 17661          ddi_xbuf_qstrategy:entry
0 17662          ddi_xbuf_qstrategy:return
0 17649          xbuf_iostart:entry
0 17341          sd_xbuf_strategy:entry
0 17385          sd_xbuf_init:entry
0 17386          sd_xbuf_init:return
0 17342          sd_xbuf_strategy:return
0 17177          sd_mapblockaddr_iostart:entry
0 17178          sd_mapblockaddr_iostart:return
0 17179          sd_pm_iostart:entry
0 17365          sd_pm_entry:entry
0 17366          sd_pm_entry:return
0 17180          sd_pm_iostart:return
0 17181          sd_core_iostart:entry
0 17407          sd_add_buf_to_waitq:entry
...

```

この例の出力では、デフォルトアクションによって、プローブが起動した CPU、DTrace によって割り当てられたプローブ ID (整数値)、プローブが起動した関数、およびプローブ名が表示されています。

例 2-7 プロバイダを指定してプローブを有効にする

```

# dtrace -P syscall
dtrace: description 'syscall' matched 452 probes
CPU    ID          FUNCTION:NAME
0      99          ioctl:return
0      98          ioctl:entry
0      99          ioctl:return
0      98          ioctl:entry
0      99          ioctl:return
0      234         sysconfig:entry
0      235         sysconfig:return
0      234         sysconfig:entry
0      235         sysconfig:return
0      168         sigaction:entry
0      169         sigaction:return
0      168         sigaction:entry
0      169         sigaction:return
0      98          ioctl:entry
0      99          ioctl:return
0      234         sysconfig:entry
0      235         sysconfig:return
0      38          brk:entry
0      39          brk:return

```

例 2-7 プロバイダを指定してプローブを有効にする (続き)

...

例 2-8 名前を指定してプローブを有効にする

```
# dtrace -n zfod
dtrace: description 'zfod' matched 3 probes
CPU    ID                FUNCTION:NAME
  0    4080                anon_zero:zfod
  0    4080                anon_zero:zfod
^C
```

例 2-9 名前を完全指定してプローブを有効にする

```
# dtrace -n clock:entry
dtrace: description 'clock:entry' matched 1 probe
CPU    ID                FUNCTION:NAME
  0    4198                clock:entry
^C
```

## DTrace アクションの基本

DTrace は、アクションによって、DTrace フレームワーク外部のシステムと対話します。もっとも一般的なアクションは、DTrace バッファヘデータを記録するアクションです。そのほかに、現在のプロセスを停止するアクション、現在のプロセス内で特定のシグナルを発生させるアクション、トレースを中断するアクションなどがあります。システム状態を変更するアクションは、「破壊アクション」と見なされます。データ記録アクションは、デフォルトで、「主バッファ」にデータを記録します。主バッファは、DTrace の呼び出し時に必ず使用されます。また、常に CPU 単位で割り当てられます。-cpu オプションを指定することにより、トレースとバッファの割り当てを単一 CPU に制限できます。DTrace のバッファリングの詳細については、『Solaris 動的トレースガイド』の第 11 章「バッファとバッファリング」を参照してください。

ここからは、組み込み D 変数を使用する D 式の例を紹介していきます。次に示すのは、もっとも頻繁に使用する D 変数の一部です。

pid	現在のプロセス ID を表す変数。
execname	現在の実行可能ファイル名を表す変数。
timestamp	起動時からの経過時間をナノ秒単位で表す変数。
curthread	現在のスレッドを示す kthread_t 構造体へのポインタを表す変数。
probemod	現在のプローブのモジュール名を表す変数。
probfunc	現在のプローブの関数名を表す変数。

`probename` 現在のプローブの名前を表す変数。

D スクリプト言語のすべての組み込み変数については、33 ページの「DTrace の組み込み変数」を参照してください。

D スクリプト言語には、特定のアクションを実行する組み込み関数も用意されています。すべての組み込み関数については、『Solaris 動的トレースガイド』の第10章「アクションとサブルーチン」を参照してください。trace() 関数は、D 式の結果をトレースバッファに記録します。次に例を示します。

- trace(pid) - 現在のプロセス ID をトレースする
- trace(execname) - 現在の実行可能ファイルの名前をトレースする
- trace(curthread->t\_pri) - 現在のスレッドの t\_pri フィールドをトレースする
- trace(probefunc) - プローブの関数名をトレースする

プローブに特定のアクションを実行させるときは、次の例のように、アクション名を {} 文字で囲んで指定します。

例 2-10 プローブのアクションを指定する

```
# dtrace -n 'readch {trace(pid)}'
dtrace: description 'readch ' matched 4 probes
CPU    ID                FUNCTION:NAME
  0    4036              read:readch      2040
  0    4036              read:readch      2177
  0    4036              read:readch      2177
  0    4036              read:readch      2040
  0    4036              read:readch      2181
  0    4036              read:readch      2181
  0    4036              read:readch       7
...
```

この例で要求されるアクションは、trace(pid) です。したがって、一番右の欄にプロセス ID 番号 (PID) が出力されています。

例 2-11 実行可能ファイル名をトレースする

```
# dtrace -m 'ufs {trace(execname)}'
dtrace: description 'ufs ' matched 889 probes
CPU    ID                FUNCTION:NAME
  0    14977            ufs_lookup:entry  ls
  0    15748            ufs_iaccess:entry  ls
  0    15749            ufs_iaccess:return  ls
  0    14978            ufs_lookup:return  ls
...
  0    15007            ufs_seek:entry    utmpd
  0    15008            ufs_seek:return    utmpd
```

例 2-11 実行可能ファイル名をトレースする (続き)

```
0 14963                ufs_close:entry      utmpd
^C
```

例 2-12 システムコールの開始時刻をトレースする

```
# dtrace -n 'syscall:::entry {trace(timestamp)}'
dtrace: description 'syscall:::entry ' matched 226 probes
CPU    ID                FUNCTION:NAME
0      312                portfs:entry         157088479572713
0      98                 ioctl:entry          157088479637542
0      98                 ioctl:entry          157088479674339
0      234                sysconfig:entry      157088479767243
...
0      98                 ioctl:entry          157088481033225
0      60                 fstat:entry          157088481050686
0      60                 fstat:entry          157088481074680
^C
```

例 2-13 複数のアクションを指定する

複数のアクションを指定する場合は、各アクションを ; 文字で区切って指定します。

```
# dtrace -n 'zfod {trace(pid);trace(execname)}'
dtrace: description 'zfod ' matched 3 probes
CPU    ID                FUNCTION:NAME
0      4080                anon_zero:zfod       2195   dtrace
0      4080                anon_zero:zfod       2195   dtrace
0      4080                anon_zero:zfod       2195   dtrace
0      4080                anon_zero:zfod       2195   dtrace
0      4080                anon_zero:zfod       2195   dtrace
0      4080                anon_zero:zfod       2197   bash
0      4080                anon_zero:zfod       2207   vi
0      4080                anon_zero:zfod       2207   vi
...
```

## データ記録アクション

この節では、デフォルトで主バッファにデータを記録するアクションを紹介します。これらのアクションを使って、投機バッファにデータを記録することもできます。投機バッファについては、[56 ページの「投機トレース」](#)を参照してください。

## trace() 関数

```
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、指定バッファに結果をトレースする `trace()` アクションです。

## tracemem() 関数

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

`tracemem()` アクションは、メモリー内のアドレスからバッファにデータをコピーします。*nbytes* には、このアクションによってコピーされるバイト数を指定します。*address* には、コピーされるデータのアドレスを D 式で指定します。指定バッファにコピーします。

## printf() 関数

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

`printf()` アクションは、`trace()` アクションと同じように D 式をトレースします。ただし、`printf()` アクションでは、`printf(3C)` 関数と同様の方法で書式を制御できます。`printf` 関数の場合と同じく、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` コマンドの出力書式が設定されます。

`printf()` アクションの詳細については、『Solaris 動的トレースガイド』の第 12 章「出力書式」を参照してください。

## printa() 関数

```
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、『Solaris 動的トレースガイド』の第 9 章「集積体」を参照してください。*format* 値を省略した場合、`printa()` アクションは DTrace コンシューマへの指令だけをトレースします。その指令を受け取ったコンシューマは、集積体をデフォルトの書式に編集して表示します。`printa()` の書式文字列については、『Solaris 動的トレースガイド』の第 12 章「出力書式」を参照してください。

## stack() 関数

```
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。`nframes` には、カーネルスタックの深さを指定します。`nframes` 値を省略した場合、`stack()` アクションは、`stackframes` オプションで指定された数のスタックフレームを記録します。

## ustack() 関数

```
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。`nframes` には、ユーザースタックの深さを指定します。`nframes` 値を省略した場合、`ustack` アクションは、`ustackframes` オプションで指定された数のスタックフレームを記録します。`ustack()` アクションは、プローブが起動するときに、呼び出しフレームのアドレスを特定します。スタックフレームは、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たときに、シンボルに翻訳されます。`strsize` にゼロ以外の値を指定した場合、`ustack()` アクションは指定された容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。

## jstack() 関数

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

`jstack()` アクションは、`ustack()` アクションの別名で、スタックフレーム数については `jstackframes` オプションに指定されている値を使用します。`jstack` アクションは、`jstackstrsize` オプションに指定された値を使って、文字列空間のサイズを決定します。デフォルトでは、`jstacksize` アクションはゼロ以外の値になります。

## 破壊アクション

破壊アクションを使用するためには、明示的な方法で有効にする必要があります。`-w` オプションを使用すれば、破壊アクションを有効にできます。破壊アクションを明示的に有効にしないまま、`dtrace` 内でその使用を試みると、`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
```

破壊アクションをはじめとする DTrace アクションについては、『Solaris 動的トレースガイド』の第 10 章「アクションとサブルーチン」を参照してください。

## 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 のセキュリティ権限については、『Solaris 動的トレースガイド』の第 35 章「セキュリティ」を参照してください。

### stop() 関数

プローブの起動によって `stop()` アクションが有効にされると、そのプローブを起動したプロセスはカーネルを出るときに停止します。このプロセスは、`proc(4)` アクションを使用してプロセスを停止したときと同じように停止します。

### raise() 関数

```
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。

### copyout() 関数

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

`copyout()` アクションは、バッファからメモリー内のアドレスへデータをコピーします。`nbytes` には、このアクションによってコピーされるバイト数を指定します。`buf` には、データのコピー元バッファを指定します。`addr` には、データのコピー先のアドレスを指定します。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にあるアドレスを指定してください。

### copyoutstr() 関数

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

`copyoutstr()` アクションは、文字列をメモリー内のアドレスにコピーします。`str` には、コピーする文字列を指定します。`addr` には、文字列のコピー先のアドレスを指定します。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にあるアドレスを指定してください。

### system() 関数

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

`system()` アクションは、`program` で指定されたプログラムを、シェルに入力として渡されたときのようにして実行します。

## 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慎重に使用してください。これらのアクションは、システムで実行中のすべてのプロセスに影響を及ぼします。さらに、そのシステムのネットワークサービスを使用しているその他のシステムに影響を及ぼすこともあります。

### breakpoint() 関数

```
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。

### panic() 関数

```
void panic(void)
```

`panic()` アクションが指定されているプローブが起動すると、カーネルパニックが発生します。このアクションを実行すると、必要に応じて強制的にシステムのクラッシュダンプを出力できます。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、システムの問題を診断できます。詳細は、『Solaris 動的トレースガイド』の第 11 章「バッファとバッファリング」と『Solaris 動的トレースガイド』の第 37 章「事後トレース」を参照してください。

### chill() 関数

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

`chill()` アクションが指定されているプローブが起動すると、指定された時間(ナノ秒)だけ DTrace が実行され続けます。`chill()` アクションは、タイミング関連の問題の調査に役立ちます。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、ディスパッチ遅延が発生します。

## DTrace 集積体

パフォーマンス関連の問題を調査するときは、通常、個々のデータポイントではなく、集積データを調べるほうが効果的です。DTrace には、いくつかの組み込みの集積関数が用意されています。データ集合のサブセットに集積関数を適用し、これらのサブセットの分析結果に再度集積関数を適用すると、データ集合全体に集積関数を適用した場合と同じ結果になります。



DTrace 機能は、集積のため、データ項目のランニングカウントを格納します。集積関数は、現在の中間結果と、関数の適用先の新規要素のみを格納します。中間結果は、CPU 単位で割り当てられます。この割り当て方式では、ロックは必要ありません。したがって、実装は本質的にスケーラブルです。

## DTrace 集積体の構文

DTrace 集積体の一般構文は、次のとおりです。

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

この一般構文では、変数は次のように定義されています。

- name** 集積体の名前。先頭に@文字が付きます。
- keys** D 式をコンマで区切って指定します。
- aggfunc** DTrace 集積関数のいずれか。
- args** その集積関数に対応する引数をコンマで区切って指定します。

表 2-1 DTrace の集積関数

関数名	引数	結果
count	なし	count 関数が呼び出される回数。
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。この集積関数は、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度数分布。この集積関数は、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。

### 例 2-14 集積関数の使用法

次に示すのは、count 集積関数を使って、プロセスごとに write(2) システムコールの数をカウントする例です。この集積体は、dtrace コマンドが終了するまで、一切のデータを出力しません。出力される内容は、dtrace コマンドがアクティブであったときに収集されたデータの概要です。

## 例 2-14 集積関数の使用法 (続き)

```
# cat writes.d
#!/usr/sbin/dtrace -s
syscall::write:entry
{
    @numWrites[execname] = count();
}

# ./writes.d
dtrace: script 'writes.d' matched 1 probe
^C
dtrace                1
date                  1
bash                   3
grep                   20
file                   197
ls                     201
```

# D 言語を使ったスクリプトの作成

---

この章では、D 言語を使って独自のスクリプトを作成するときに必要となる基本情報を提供します。

## D スクリプトの作成

DTrace プローブを複雑に組み合わせると、コマンド行での使用が難しくなる場合もあります。dtrace コマンドは、スクリプトをサポートしています。dtrace コマンドに `-s` オプションとスクリプトファイル名を指定することで、スクリプトを指定できます。実行可能な DTrace インタプリタファイルも作成できます。DTrace インタプリタファイルは、常に `#!/usr/sbin/dtrace -s` から始まります。

## 実行可能な D スクリプト

次のサンプルスクリプト `syscall.d` は、実行可能ファイルがシステムコールを開始するたびに、その実行可能ファイルの名前をトレースします。

```
syscall:::entry
{
    trace(execname);
}
```

ファイル名は接尾辞 `.d` で終わっています。D スクリプトの末尾は、通常、この形式になります。このスクリプトは、DTrace コマンド行から次のコマンドを使用して実行できます。

```
# dtrace -s syscall.d
dtrace: description 'syscall' matched 226 probes
CPU   ID                FUNCTION:NAME
  0    312                pollsys:entry   java
  0    98                 ioctl:entry     dtrace
```

```

0    98                ioctl:entry    dtrace
0    234               sysconfig:entry  dtrace
0    234               sysconfig:entry  dtrace
0    168               sigaction:entry  dtrace
0    168               sigaction:entry  dtrace
0    98                ioctl:entry    dtrace
^C

```

このスクリプトは、次の2つの手順でコマンド行にファイル名を指定する方法で実行できます。まず、ファイルの先頭行がインタプリタの呼び出しになっていることを確認します。インタプリタの呼び出し行は、`#!/usr/sbin/dtrace -s` です。次に、ファイルの実行権を設定します。

例 3-1 コマンド行から D スクリプトを実行する

```

# cat syscall.d
#!/usr/sbin/dtrace -s

syscall:::entry
{
    trace(execname);
}

# chmod +x syscall.d
# ls -l syscall.d
-rwxr-xr-x  1 root    other      62 May 12 11:30 syscall.d
# ./syscall.d
dtrace: script './syscall.d' matched 226 probes
CPU    ID                FUNCTION:NAME
0      98                ioctl:entry    dtrace
0      98                ioctl:entry    dtrace
0      312               pollsys:entry  java
0      312               pollsys:entry  java
0      312               pollsys:entry  java
0      98                ioctl:entry    dtrace
0      98                ioctl:entry    dtrace
0      234               sysconfig:entry  dtrace
0      234               sysconfig:entry  dtrace
^C

```

## D リテラル文字列

D 言語はリテラル文字列をサポートします。DTrace の文字列は、NULL バイトで終了す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。DTrace 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、各文字列を固定サイズの配列に格納します。文字列の長さは、あらかじめ定義された文字列制限長を超えることはできません。この制限は 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、『Solaris 動的トレースガイド』の第 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。D リテラル文字列については、『Solaris 動的トレースガイド』の第 6 章「文字列」を参照してください。

例 3-2 `trace()` 関数で D リテラル文字列を使用する

```
# cat string.d

#!/usr/sbin/dtrace -s

fbt::bdev_strategy:entry
{
    trace(execname);
    trace(" is initiating a disk I/O\n");
}
```

このリテラル文字列の末尾の記号 `\n` によって、新規の行が生成されます。このスクリプトを実行するには、次のコマンドを入力します。

```
# dtrace -s string.d
dtrace: script 'string.d' matched 1 probes
CPU    ID                FUNCTION:NAME
  0    9215                bdev_strategy:entry  bash is initiating a disk I/O
  0    9215                bdev_strategy:entry  vi is initiating a disk I/O
  0    9215                bdev_strategy:entry  vi is initiating a disk I/O
  0    9215                bdev_strategy:entry  sched is initiating a disk I/O
```

^C

`dtrace` コマンドを `-q` オプション付きで実行した場合、スクリプト内またはコマンド行呼び出しに明示的に指定されたアクションだけが記録されます。`dtrace` コマンドが通常生成するデフォルト出力は抑制されます。

```
# dtrace -q -s string.d
ls is initiating a disk I/O
cat is initiating a disk I/O
fsflush is initiating a disk I/O
vi is initiating a disk I/O
^C
```

## 引数を使用するDスクリプトを作成する

dtrace コマンドを使って、実行可能なインタプリタファイルを作成できます。このファイルには、実行権を設定する必要があります。ファイルの先頭行は、`#!/usr/sbin/dtrace -s` にする必要があります。この行には、dtrace コマンドのほかのオプションを指定できます。複数のオプションを指定する場合も、ダッシュ(-)は1つだけ入力してください。s オプションは、次の例のように、最後に指定してください。

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

dtrace コマンドのオプションを、Dスクリプトの `#pragma` 行を使用して指定できます。次のDスクリプトの抜粋を参照してください。

```
# cat -n mem2.d
1  #!/usr/sbin/dtrace -s
2
3  #pragma D option quiet
4  #pragma D option verbose
5
6  vminfo:::
...

```

次の表に、`#pragma` 行で使用できるオプション名を一覧します。

表 3-1 DTrace コンシューマオプション

オプション名	値	dtrace 別名	説明
aggrate	時間		集積体の読み取りレート
aggsz	サイズ		集積体バッファサイズ
bufresz	auto または manual		バッファのサイズ変更ポリシー
bufsz	サイズ	-b	主バッファサイズ
cleanrate	時間		クリーンアップレート
cpu	スカラー	-c	トレースを有効にする CPU
defaultargs	—		未知のマクロ引数の参照を許可する
destructive	—	-w	破壊アクションを許可する

表 3-1 DTrace コンシューマオプション (続き)

オプション名	値	dtrace 別名	説明
dynvarsize	サイズ		動的可変変数空間のサイズ
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
grabanon	—	-a	匿名状態を要求する
jstackframes	スカラー		jstack() のデフォルトスタックフレームの数
jstackstrsize	スカラー		jstack() の文字列空間のデフォルトサイズ
nspec	スカラー		投機の数
quiet	—	-q	明示的にトレースされたデータだけを出力する
speccsize	サイズ		投機バッファサイズ
strsize	サイズ		文字列サイズ
stackframes	スカラー		スタックフレームの数
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
statusrate	時間		状態チェックレート
switchrate	時間		バッファ切り替えレート
ustackframes	スカラー		ユーザースタックフレームの数

一連の組み込みマクロ変数も、D スクリプトで参照できます。これらのマクロ変数は、D コンパイラで定義されます。

\$[0-9]+    マクロ引数  
 \$egid    実効グループ ID  
 \$euid    実効ユーザー ID  
 \$gid    実グループ ID

\$pid	プロセス ID
\$pgid	プロセスグループ ID
\$ppid	親プロセス ID
\$projid	プロジェクト ID
\$sid	セッション ID
\$target	ターゲットプロセス ID
\$taskid	タスク ID
\$uid	実ユーザー ID

### 例 3-3 PID 引数の例

次の例では、D スクリプト `syscalls2.d` に、実行中の `vi` プロセスの PID が渡されます。`vi` コマンドが終了すると、D スクリプトも終了します。

```
# cat -n syscalls2.d
 1  #!/usr/sbin/dtrace -qs
 2
 3  syscall::entry
 4  /pid == $1/
 5  {
 6    @[probefunc] = count();
 7  }
 8  syscall::rexit:entry
 9  {
10    exit(0);
11  }
```

```
# pgrep vi
```

```
2208
```

```
# ./syscalls2.d 2208
```

rexit	1
setpgrp	1
creat	1
getpid	1
open	1
lstat64	1
stat64	1
fdsync	1
unlink	1
close	1
alarm	1
lseek	1



## 例 3-3 PID 引数の例 (続き)

sigaction	1
ioctl	1
read	1
write	1

## DTrace の組み込み変数

次に、DTrace フレームワークのすべての組み込み変数を一覧します。

<code>int64_t arg0, ..., arg9</code>	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。
<code>args[]</code>	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)。 <code>args[]</code> 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、 <code>read(2)</code> システムコールプローブで <code>args[]</code> を参照する場合、 <code>args[0]</code> の型は <code>int</code> 、 <code>args[1]</code> の型は <code>void *</code> 、 <code>args[2]</code> の型は <code>size_t</code> になります。
<code>uintptr_t caller</code>	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。
<code>chipid_t chip</code>	現在の物理チップの CPU チップ識別子です。詳細は、『Solaris 動的トレースガイド』の第 26 章「sched プロバイダ」を参照してください。
<code>processorid_t cpu</code>	現在の CPU の CPU 識別子です。詳細は、『Solaris 動的トレースガイド』の第 26 章「sched プロバイダ」を参照してください。
<code>cpuinfo_t *curcpu</code>	現在の CPU の CPU 情報です。詳細は、『Solaris 動的トレースガイド』の第 26 章「sched プロバイダ」を参照してください。
<code>lwpsinfo_t *curlwpsinfo</code>	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、 <code>proc(4)</code> のマニュアルページに記載されています。
<code>psinfo_t *curpsinfo</code>	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、 <code>proc(4)</code> のマニュアルページに記載されています。
<code>kthread_t *curthread</code>	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 <code>kthread_t</code> のアドレスです。 <code>kthread_t</code> は

---

	<p>&lt;sys/thread.h&gt; 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。</p>
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。
lgrp_id_t lgrp	現在の CPU をメンバーに持つ遅延グループの近傍性グループ ID です。DTrace での CPU 管理については、『Solaris 動的トレースガイド』の第 26 章「sched プロバイダ」を参照してください。近傍性グループについては、『プログラミングインタフェース』の第 4 章「近傍性グループ API」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部分です。
string probeprovider	現在のプローブの記述に含まれるプロバイダ名部分です。
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、『Solaris 動的トレースガイド』の第 26 章「sched プロバイダ」を参照してください。

---

<code>string root</code>	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。
<code>uint_t stackdepth</code>	現在のスレッドのプロープ起動時のスタックフレームの深さを表します。
<code>id_t tid</code>	現在のスレッドのスレッド ID です。スレッドにユーザープロセスが関連付けられている場合、この値は、 <code>pthread_self(3C)</code> の呼び出し結果と等しくなります。
<code>uint64_t timestamp</code>	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。
<code>uid_t uid</code>	現在のプロセスの実ユーザー ID です。
<code>uint64_t uregs[]</code>	現在のスレッドの、プロープ起動時のユーザーモード登録値(保存済み)です。 <code>uregs[]</code> 配列の使用方法については、『Solaris 動的トレースガイド』の第 33 章「ユーザープロセスのトレース」を参照してください。
<code>uint64_t vtimestamp</code>	ナノ秒タイムスタンプカウンタの現在の値です。このカウンタは、現在のスレッドの CPU 上での実行時間を示します。これには、DTrace の述語やアクションの実行にかかる時間は含まれません。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。
<code>uint64_t walltimestamp</code>	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。



# DTrace の使用法

---

この章では、DTrace を使って、よくある基本的なタスクを実行する方法を確認します。さらに、何種類かのトレースについても説明します。

## パフォーマンス監視

いくつかの DTrace プロバイダは、既存のパフォーマンス監視ツールに対応するプローブを実装しています。

- `vminfo` プロバイダ - `vmstat(1M)` ツールに対応するプローブを実装
- `sysinfo` プロバイダ - `mpstat(1M)` ツールに対応するプローブを実装
- `io` プロバイダ - `iostat(1M)` ツールに対応するプローブを実装
- `syscall` プロバイダ - `truss(1)` ツールに対応するプローブを実装

DTrace 機能を使用すると、バンドルされているツールから得られるのと同じ情報を、より柔軟に抽出できます。DTrace 機能は、プローブの起動時に使用できる任意のカーネル情報を提供します。DTrace 機能を使って、プロセス ID、スレッド ID、スタックトレースなどの情報を受け取ることができます。

## sysinfo プロバイダを使ってパフォーマンスの問題を検査する

`sysinfo` プロバイダは、`sys` 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` などのシステム監視ユーティリティの入力となります。`sysinfo` プロバイダのプローブは、`kstat` コマンドで表示される `sys` の値が増分される直前に起動します。次に、`sysinfo` プロバイダが提供するプローブを一覧します。

`bawrite`                      バッファ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。
bwrite	バッファ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。
cpu_ticks_idle	定期的なシステムクロックにより、「CPUがアイドル状態である」という判断が下されたとき起動するプローブ。このプローブは、システムクロックのコンテキストで起動します。したがって、システムクロックを実行しているCPU上で起動することになります。cpu_tの引数(arg2)は、アイドル状態だと判断されたCPUを表します。
cpu_ticks_kernel	定期的なシステムクロックにより、「CPUがカーネルで実行中である」という判断が下されたとき起動するプローブ。このプローブは、システムクロックのコンテキストで起動します。したがって、システムクロックを実行しているCPU上で起動することになります。cpu_tの引数(arg2)は、カーネルで実行中であると判断されたCPUを表します。
cpu_ticks_user	定期的なシステムクロックにより、「CPUがユーザーモードで実行中である」という判断が下されたとき起動するプローブ。このプローブは、システムクロックのコンテキストで起動します。したがって、システムクロックを実行しているCPU上で起動することになります。cpu_tの引数(arg2)は、ユーザーモードで実行中であると判断されたCPUを表します。
cpu_ticks_wait	定期的なシステムクロックにより、「CPU上に入出力待ちのスレッドがあるほかはアイドル状態である」という判断が下されたとき起動するプローブ。このプローブは、システムクロックのコンテキストで起動します。したがって、システムクロックを実行しているCPU上で起動することになります。cpu_tの引数(arg2)は、入出力待ち状態だと判断されたCPUを表します。
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_swthc	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。

<code>modunload</code>	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。
<code>msg</code>	<code>msgsnd(2)</code> または <code>msgrcv(2)</code> 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。
<code>mutex_adenters</code>	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、 <code>lockstat</code> プロバイダの <code>adaptive-block</code> プローブ、または <code>adaptive-spin</code> プローブも起動します。
<code>namei</code>	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。
<code>nthreads</code>	スレッドが作成されたときに起動するプローブ。
<code>phread</code>	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。
<code>phwrite</code>	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。
<code>procovf</code>	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。
<code>pswitch</code>	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。
<code>readch</code>	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。読み取りは、 <code>read(2)</code> 、 <code>readv(2)</code> 、または <code>pread(2)</code> システムコールによって行われます。 <code>arg0</code> には、正常に読み取られたバイト数が格納されます。
<code>rw_rdfails</code>	書き込み側が読み取りロックまたは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、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、 <code>lockstat</code> プロバイダの <code>rw-block</code> プローブも起動します。
<code>rw_wrfails</code>	読み取り側か別の書き込み側が読み取りロックまたは書き込みロックを保持している場合に、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、 <code>lockstat</code> プロバイダの <code>rw-block</code> プローブも起動します。
<code>sema</code>	システムコール <code>semop(2)</code> が発行されたあと、セマフォ操作が行われる前に起動するプローブ。
<code>sysexec</code>	システムコール <code>exec(2)</code> が発行されたときに起動するプローブ。
<code>sysfork</code>	システムコール <code>fork(2)</code> が発行されたときに起動するプローブ。
<code>sysread</code>	システムコール <code>read</code> 、 <code>readv</code> 、または <code>pread</code> が発行されたときに起動するプローブ。
<code>sysvfork</code>	システムコール <code>vfork(2)</code> が発行されたときに起動するプローブ。
<code>syswrite</code>	システムコール <code>write(2)</code> 、 <code>writv(2)</code> 、または <code>pwrite(2)</code> 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はこのプローブを起動しないことがあります。
ufsdireblk	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。
ufsiopage	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に起動するプローブ。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。
wait_ticks_io	定期的なシステムクロックにより、「CPU 上に入出力待ちのスレッドがあるほかはアイドル状態である」という判断が下されたときに起動するプローブ。このプローブは、システムクロックのコンテキストで起動します。したがって、システムクロックを実行している CPU 上で起動することになります。cpu_t の引数 (arg2) は、入出力待ち状態だと判断された CPU を表します。wait_ticks_io と cpu_ticks_wait には、意味上の違いはありません。wait_ticks_io は、履歴上の理由で提供されているだけです。
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、writev、pwrite 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。

#### 例 4-1 sysinfo プローブで quantize 集積関数を使用する

quantize 集積関数は、引数の二乗度数分布を示す棒グラフを表示します。次の例では、quantize 関数を使って、10 秒間にシステム上の全プロセスで実行される read 呼び出しのサイズを確認します。sysinfo プローブの引数 arg0 には、統計情報の増分の分量を指定します。ほとんどの sysinfo プローブでは、この値は 1 です。ただし、readch プローブと writech プローブは例外です。これらのプローブでは、引数 arg0 に、実際に読み取られるバイト数、または実際に書き込まれるバイト数を指定します。



例 4-1 sysinfo プローブで quantize 集積関数を使用する (続き)

```
# cat -n read.d
1  #!/usr/sbin/dtrace -s
2  sysinfo:::readch
3  {
4      @[execname] = quantize(arg0);
5  }
6
7  tick-10sec
8  {
9      exit(0);
10 }
```

```
# dtrace -s read.d
dtrace: script 'read.d' matched 5 probes
CPU      ID                FUNCTION:NAME
0        36754              :tick-10sec
```

```
bash
value  ----- Distribution ----- count
0      |                                0
1      |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 13
2      |                                0
```

```
file
value  ----- Distribution ----- count
-1     |                                0
0      |                                2
1      |                                0
2      |                                0
4      |                                6
8      |                                0
16     |                                0
32     |                                6
64     |                                6
128    |@@                                16
256    |@@@@                              30
512    |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 199
1024   |                                0
2048   |                                0
4096   |                                1
8192   |                                1
16384  |                                0
```

```
grep
value  ----- Distribution ----- count
```

## 例4-1 sysinfo プローブで quantize 集積関数を使用する (続き)

```

-1 | 0
0 | @@@@@@@@@@@@@@@@@@@@@@ 99
1 | 0
2 | 0
4 | 0
8 | 0
16 | 0
32 | 0
64 | 0
128 | 1
256 | @@@@ 25
512 | @@@@ 23
1024 | @@@@ 24
2048 | @@@@ 22
4096 | 4
8192 | 3
16384 | 0

```

## 例4-2 クロスコールのソースを確認する

次に、mpstat(1M) コマンドの出力例を示します。

```

CPU minf mjf xcal intr ithr csw icsw migr smtx srw syscl usr sys wt idl
0 2189 0 1302 14 1 215 12 54 28 0 12995 13 14 0 73
1 3385 0 1137 218 104 195 13 58 33 0 14486 19 15 0 66
2 1918 0 1039 12 1 226 15 49 22 0 13251 13 12 0 75
3 2430 0 1284 220 113 201 10 50 26 0 13926 10 15 0 75

```

xcal 欄と syscl 欄の値が異常に大きいため、システムパフォーマンスが浪費されている可能性があります。システムは比較的アイドル状態にあり、入出力の待ち時間が異常に長くなっているわけでもありません。xcal 欄の数値は sys kstat の xcalls フィールドから読み取られたものであり、秒単位で表されています。クロスコールを行う実行可能ファイルを確認するには、次のような dtrace コマンドを入力します。

```

# dtrace -n 'xcalls {@[execname] = count()}'
dtrace: description 'xcalls ' matched 3 probes
^C
  find                2
  cut                  2
  snmpd                2
  mpstat               22
  sendmail             101
  grep                 123
  bash                 175

```

## 例 4-2 クロスコールのソースを確認する (続き)

```

dtrace                435
sched                 784
xargs                 22308
file                  89889
#

```

出力結果から、大量のクロスコールが file(1) と xargs(1) のプロセスによって行われていることがわかります。これらのプロセスを確認するには、pgrep(1) コマンドと ptree(1) を使用します。

```

# pgrep xargs
15973
# ptree 15973
204  /usr/sbin/inetd -s
    5650 in.telnetd
        5653 -sh
            5657 bash
                15970 /bin/sh ./findtxt configuration
                    15971 cut -f1 -d:
                        15973 xargs file
                            16686 file /usr/bin/tbl /usr/bin/troff /usr/bin/ul /usr/bin/vgrind /usr/bin/catman

```

出力結果から、xargs コマンドと file コマンドがカスタムユーザーシェルスクリプトの一部であることがわかります。このスクリプトを検出するには、次のコマンドを実行します。

```

# find / -name findtxt
/usrsl/james/findtxt
# cat /usrsl/james/findtxt
#!/bin/sh
find / -type f | xargs file | grep text | cut -f1 -d: > /tmp/findtxt$$
cat /tmp/findtxt$$ | xargs grep $1
rm /tmp/findtxt$$
#

```

このスクリプトは、多くのプロセスを同時に実行しています。パイプ経由で大量のプロセス間通信が行われています。パイプ数が多いと、スクリプトリソースが集中的に使用されます。このスクリプトは、システム上のすべてのテキストファイルの検出を試みたあと、各ファイルから特定のテキストを検索します。

## ユーザープロセスをトレースする

この節では、ユーザープロセスアクティビティのトレースに役立つ DTrace 機能に注目し、例を挙げながらその用法について説明します。

### サブルーチン `copyin()` と `copyinstr()` を使用する

DTrace プローブは、Solaris カーネル内で実行されます。プローブは、サブルーチン `copyin()` または `copyinstr()` を使って、ユーザープロセスのデータをカーネルのアドレス空間へコピーします。

たとえば、次のような `write()` 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` 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。次に、正しくない D プログラムの例を示します。

```
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、システムコールを実行しているプロセス内のメモリーを参照するアドレスを指定します。サブルーチン `copyinstr()` を使って、このアドレスの文字列を読み取ります。`printf()` アクションを使って、結果を記録します。

```
syscall::write:entry
{
    printf("%s", copyinstr(arg1)); /* correct use of arg1 */
}
```

このスクリプトの出力から、`write` 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。

### エラーの回避

サブルーチン `copyin()` と `copyinstr()` では、一度も使ったことがないユーザーアドレスからの読み取りは実行できません。アドレスが含まれているページが、過去に一度もアクセス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
```

前の例の出力では、このアプリケーションが正常に動作していて、`arg0` に指定されたアドレスは有効です。しかし、`arg0` に指定されたアドレスは、対応するプロセスがまだ一度もアクセスしたことがないページを参照しています。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、データの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 の干渉を排除する

`write` 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せん。`write()` 関数の呼び出しのたびに、`dtrace` コマンドは、出力表示のため `write()` 関数を呼び出します。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。この動作を防ぐには、次の例のような述語を使用します。

```
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。述語 `/pid != $pid/` を使用すれば、このスクリプトの実行に関連したすべての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。`prstat(1M)` コマンドを使用すると、プロセスの動作を確認できます。

```
$ 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

このレポートから、write() 関数へのシステムコールが多数存在することがわかります。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(); }'
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

この出力は、このプロセスが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ることを示しています。

## ustack() アクション

`ustack()` アクションは、ユーザーのスレッドのスタックをトレースします。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()` 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` コマンド行に最初に指定された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()` アクションは、スタックのプログラムカウンタ (PC) の値を記録します。すると、`dtrace` コマンドにより、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` コマンドが、PC 値を解決できない場合は、16 進整数として出力します。

`ustack()` データに出力書式が設定される前にプロセスが終了したり、強制終了されたりすると、`dtrace` コマンドは、スタックトレース内の PC 値をシンボル名に変換できなくなる可能性があります。この場合、`dtrace` コマンドは、これらの値を 16 進整数として表示します。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。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。次の例では、`open` システムコールプローブを使用しています。この方法は、`ustack` アクションを使用するあらゆるスクリプトで使用できます。

```
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。動的な変数がクリアされたら、`stop_pids[pid]` の値をゼロ (0) に設定します。

## 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。

## 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。マクロ変数 `$1` は、コマンド行内の最初のオペランドに展開されます。このマクロ変数は、トレース対象のプロセスのプロセス ID と同じです。マクロ変数 `$2` は、コマンド行内の 2 番目のオペランドに展開されます。このマクロ変数は関数名になっています。すべての関数呼び出しは、この関数からトレースされます。



例 4-3 userfunc.d: ユーザー関数の開始 (entry) と終了 (return) をトレースする

```
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。dtrace 機能を使って重要なプロセスを作成し、計測するときには、\$target マクロ変数と dtrace の -c

オプションおよび `-p` オプションを使用できます。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` 関数呼び出しの内訳がわかります。

```
pid$target:libc.so::entry
{
    @[probefunc] = count();
}
```

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、対応する命令の関数内でのオフセット (16 進整数) です。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するには、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:
```

次の例は、pidプロバイダと投機トレースを組み合わせて、関数内のすべての命令をトレースする方法を示しています。

例4-4 errorpath.d: ユーザー関数呼び出しのエラーパスをトレースする

```
pid$1::$2:entry
{
    self->spec = speculation();
    speculate(self->spec);
    printf("%x %x %x %x %x", arg0, arg1, arg2, arg3, arg4);
}

pid$1::$2:
/self->spec/
{
    speculate(self->spec);
}

pid$1::$2:return
/self->spec && arg1 == 0/
{
    discard(self->spec);
    self->spec = 0;
}

pid$1::$2:return
/self->spec && arg1 != 0/
{
    commit(self->spec);
    self->spec = 0;
}
```

errorpath.d を実行すると、このスクリプトの出力は次のようになります。

```
# ./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
```

## 匿名トレース

この節では、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。匿名有効化を作成できるのは、スーパーユーザーだけです。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` コマンドを `-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` コマンドは、`dtrace(7D)` ドライバの構成ファイルに、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。構成ファイルは通常 `/kernel/drv/dtrace.conf` です。これらのプロパティは、`dtrace` ドライバのロード時にこのドライバによって読み取られます。このドライバは、指定されたプローブと指定されたアクションを有効にし、「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` 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ともに、必要に応じてロードされます。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` ドライバをロードする必要があります。`dtrace` コマンドは、必要な `dtrace` プロバイダと `dtrace` ドライバの `/etc/system` (`system(4)` 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` ドライバから、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、`dtrace` コマンドの通常使用時に使用できる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せずに `dtrace -A` コマンドを実行します。

## 匿名状態を要求する

マシンが完全に起動したら、`dtrace` コマンドに `-a` オプションを指定して、既存の匿名状態を要求できます。デフォルトでは、`-a` オプションは、匿名状態を要求し、既存のデータを処理したあと、実行を継続します。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。

匿名状態がカーネルから消費された場合、匿名状態を元に戻すことはできません。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 コマンドは次の例のようなメッセージを返します。

```
dtrace: could not enable tracing: No anonymous tracing state
```

欠落やエラーが発生した場合、dtrace コマンドは、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

次の例では、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 ドライバ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 コマンドに -a オプションを指定すると、匿名状態が消費されます。

```
# 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
...

```

次の例では、`iprbattach()` から呼び出された関数だけに注目します。

```

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` ドライバ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 オプションを指定して実行します。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
...

```

## 投機トレース

この節では、DTraceの「投機トレース」について説明します。投機トレース機能では、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に「コミット」するか「破棄」するかを決めることができます。重要でないイベントを除去する手段としては、主に「述語」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。プローブの起動後まで、そのプローブイベントが重要かどうかわからない場合には、述語はあまり有用ではありません。

あるシステムコールが一般的なエラーコードを出力してときどき異常終了する場合、エラー条件の原因となっているコードパスを調べることをお勧めします。投機トレース機能を使用すれば、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、重要な出力だけがトレースデータとして保持されるため、事後処理の必要がありません。

## 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 4-1 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 の投機バッファにトレースする
commit	ID	指定された ID の投機バッファをコミットする
discard	ID	指定された ID の投機バッファを破棄する



## 投機の作成

`speculation()` 関数は、投機バッファを割り当て、投機識別子を返します。以後、`speculate()` 関数の呼び出し時には、この投機識別子を使用します。投機識別子の値がゼロの場合、この投機識別子は常に無効です。ただし、`speculate()`、`commit()`、`discard()` のいずれかの関数に渡すことは可能です。`speculation()` の呼び出しに失敗した場合、`dtrace` コマンドは、次のようなメッセージを返します。

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

## 投機の使用

投機を使用するには、すべてのデータ記録アクションの実行前に、`speculation()` から返された識別子を `speculate()` 関数に渡す節を使用してください。`speculate()` と同じ節に含まれるすべてのデータ記録アクションは、投機的にトレースされます。1つのDプロープ節内で、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、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ことができます。同じ節にこれら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。`speculate()` 関数のあとに、`speculate()` 関数を使用することはできません。1つの節で使用できる投機は1つだけです。`speculate()` 関数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(有効化されたプローブIDのみをトレースする)が投機的にトレースされます。

`speculation()` 関数を使用するときには、通常は、`speculation()` 関数の結果をスレッド固有変数に割り当てます。そのあとは、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。

### 例4-5 `speculation()` 関数の一般的な使用例

```
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。投機バッファをコミットすると、このバッファ内のデータが主バッファにコピーされます。指定した投機バッファに、主バッファ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、バッファの欠落カウンタの値が大きくなります。バッファを複数のCPU上で投機的にトレースした場合、コミット操作を実行した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()` の実行後にコピーされます。

コミットが実行されている間は、その投機バッファを後続の `speculation()` 呼び出しで使用できません。各CPUの投機バッファが対応するCPUの主バッファに完全にコピーされるまで待つ必要があります。コミット中のバッファに `speculate()` 関数呼び出しの結果を書き込もうとすると、データは破棄されます。このとき、エラーは生成されません。その後の `commit()` や `discard()` の呼び出しも、エラーを返さずに失敗します。`commit()` 関数が含まれる節に、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、複数のバッファを同時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。`discard()` 関数を呼び出しているCPU上でのみ投機がアクティブにされている場合は、そのバッファを、後続の `speculation()` 関数呼び出しですぐに使用できます。複数のCPU上で投機がアクティブにされている場合、破棄されたバッファは、`discard()` の呼び出し後に、後続の `speculation()` 関数呼び出しで使用できるようになります。`speculation()` 関数を呼び出したときに、使用可能な投機バッファが存在しない場合は、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。`open()` が異常終了したときは、`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。次の例を参照してください。

例 4-6 `specopen.d`: 異常終了した `open()` のコードフロー

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

syscall::open:entry,
syscall::open64:entry
{
```

## 例 4-6 specopen.d: 異常終了した open() のコードフロー (続き)

```

/*
 * 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
 * speculation is subsequently committed.
 */
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/
{
    /*

```

例 4-6 specopen.d: 異常終了した open() のコードフロー (続き)

```

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

```

このスクリプトを実行すると、次の例のような結果が出力されます。

```

# ./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         -> lookupnpat
1         -> lookupnpvp
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         -> dnlibc_lookup
1         -> bcmap
1         <- bcmap
1         <- dnlibc_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         <- lookupnpvp
1         <- lookupnpnat
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





# 索引

---

## C

copyin(), 44  
copyinstr(), 44

## D

dtrace の介入, 45

## P

pid プロバイダ, 48, 50

## S

speculation() 関数, 57

## U

ustack(), 47

## あ

アクション

jstack, 22  
printa, 21  
printf, 21  
stack, 22  
trace, 21

tracemem, 21

ustack, 22

データ記録, 20

破壊, 22 アクション, 破壊 (続き)

breakpoint, 24

chill, 24

copyout, 23

copyoutstr, 23

panic, 24

raise, 23

stop, 23

system, 23

## か

関数境界のテスト (FBT), 48

## さ

サブルーチン

copyin(), 44

copyinstr(), 44

## し

述語, 11

て  
データ記録アクション, 20

と  
投機, 56  
    コミット, 58  
    作成, 57  
    使用法, 57  
    使用例, 58  
    破棄, 58  
匿名トレース, 52  
    使用例, 53  
    匿名状態の要求, 52  
匿名有効化, 52

は  
破壊アクション, 22  
    カーネル, 24  
    プロセス, 23

ふ  
プローブ, `syscall()`, 45

め  
命令のトレース, 50

も  
文字列, 28  
    型, 29

ゆ  
ユーザープロセスのトレース, 44

れ  
例  
    投機, 58  
    匿名トレース, 53