



Red Hat Enterprise Linux 7

SystemTap ビギナーズガイド

SystemTap 入門

Red Hat Enterprise Linux 7 SystemTap ビギナーズガイド

SystemTap 入門

Enter your first name here. Enter your surname here.

Enter your organisation's name here. Enter your organisational division here.

Enter your email address here.

法律上の通知

Copyright © 2023 | You need to change the HOLDER entity in the en-US/SystemTap_Beginners_Guide.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

このドキュメントは、プレビュー版としてのみ提供されています。これは開発中であり、大幅に変更される可能性があります。含まれる情報は不完全なものと考え、注意して使用してください。このガイドでは、SystemTap を使用して &PRODUCT;7 のさまざまなサブシステムを詳細に監視する基本的な手順を説明します。SystemTap ビギナーズガイド は、RHCSA 試験を受験したユーザー、または &PRODUCT;7 について同様のレベルの専門知識を持っているユーザーに推奨されません。

目次

第1章 はじめに	4
1.1. 本ガイドの目的	4
1.2. SYSTEMTAP の機能	4
第2章 SYSTEMTAP の使用	6
2.1. インストールと設定	6
2.1.1. SystemTap のインストール	6
2.1.2. 必要なカーネル情報パッケージのインストール	6
2.1.3. 初期テスト	7
2.2. 他のコンピューター用のインストールメンテーション生成	7
2.3. SYSTEMTAP スクリプトの実行	9
2.3.1. SystemTap フライトレコーダーモード	11
2.3.1.1. メモリー内フライトレコーダー	11
2.3.1.2. ファイルフライトレコーダー	12
第3章 SYSTEMTAP の作動方法	13
3.1. アーキテクチャー	13
3.2. SYSTEMTAP スクリプト	13
3.2.1. イベント	15
3.2.2. Systemtap ハンドラー/ボディー	17
3.3. 基本的な SYSTEMTAP ハンドラーコンストラクト	20
3.3.1. 変数	20
3.3.2. 条件付き (conditional) ステートメント	21
3.3.3. コマンドラインの引数	23
3.4. 連想アレイ	23
3.5. SYSTEMTAP でのアレイ演算	24
3.5.1. 関連する値の割り当て	24
3.5.2. アレイからの値の読み取り	24
3.5.3. 関連する値の増加	25
3.5.4. アレイ内での複数要素の処理	25
3.5.5. アレイおよびアレイ要素の消去/削除	26
3.5.6. 条件付きステートメントにおけるアレイの使用	27
3.5.7. 統計集計の計算	28
3.6. TAPSETS	30
第4章 便利な SYSTEMTAP スクリプト	31
4.1. ネットワーク	31
4.1.1. ネットワークのプロファイリング	31
4.1.2. ネットワークソケットコードで呼び出された関数の追跡	33
4.1.3. 着信 TCP 接続の監視	34
4.1.4. カーネルでのネットワークパケットドロップの監視	35
4.2. ディスク	36
4.2.1. ディスク読み取り/書き込みトラフィックの要約	36
4.2.2. ファイル読み取り/書き込みの I/O 時間の追跡	38
4.2.3. 累積 I/O の追跡	41
4.2.4. I/O 監視 (デバイスごと)	42
4.2.5. ファイルの読み取りおよび書き込みの監視	43
4.2.6. ファイル属性の変更の監視	44
4.3. プロファイリング	45
4.3.1. 関数呼び出しのカウント	45
4.3.2. 呼び出し先の追跡	46
4.3.3. カーネルおよびユーザースペースで費やした時間の判定	47

4.3.4. ポーリングアプリケーションの監視	49
4.3.5. 最もよく使われるシステムコールの追跡	52
4.3.6. プロセスごとのシステムコールボリュームの追跡	54
4.4. 競合ユーザースペースのロックの特定	56
第5章 SYSTEMTAP のエラーを理解する	58
5.1. 解析エラーとセマンティックエラー	58
5.2. ランタイムエラーおよび警告	60
第6章 リファレンス	61
付録A 更新履歴	62
索引	62

第1章 はじめに

SystemTap は、オペレーティングシステム (特にカーネル) の動作を詳細に観察および監視できる追跡およびプロベリングのツールです。**netstat**、**ps**、**top**、**iostat**などのツールの出力と同様の情報を提供します。ただし、SystemTap は、収集された情報に対してより多くのフィルタリングおよび分析オプションを提供するように設計されています。

SystemTap は、システム管理者が Red Hat Enterprise Linux 5 以降のパフォーマンス監視ツールとして使用できます。これは、他の同様のツールではシステムのボトルネックを正確に特定できず、システムアクティビティーの詳細な分析が必要な場合に最も役立ちます。同様に、アプリケーション開発者は SystemTap を使用して、Linux システム内でアプリケーションがどのように動作するかを詳細に監視することもできます。

1.1. 本ガイドの目的

SystemTap は、分析を詳細に行うために、稼働中の Linux システムを監視するインフラストラクチャーを提供します。これは、管理者や開発者がバグやパフォーマンス問題の根本的な原因を特定する手助けとなります。

SystemTap を使用しないで、実行中のカーネルのアクティビティーを監視しようとする、インストールメント化、再コンパイル、インストール、および再起動という面倒なプロセスが必要になります。SystemTap を使用するとこのプロセスが不要になり、ユーザーが作成する SystemTap スクリプトを実行するだけで同様の情報を収集できるようになります。

ただし、SystemTap は当初、Linux カーネルに関する中級から上級の知識を備えたユーザーのために設計されました。このため、SystemTap は Linux カーネルに関する知識と経験があまりない管理者や開発者にはそれほど有用なものではありません。さらに、既存の SystemTap ドキュメントのほとんどは、同様に知識と経験が豊富なユーザーを対象としています。

そこで、『SystemTap ビギナーズガイド』では初級者を対象に、以下を説明します。

- SystemTap の概要を紹介するほか、ユーザーにアーキテクチャーについて説明し、全カーネルタイプの設定手順を提供します。
- システムの異なるコンポーネントで詳細なアクティビティーを監視するための事前作成された SystemTap スクリプトを提供し、それらの実行方法と出力の分析方法も提供します。

1.2. SYSTEMTAP の機能

SystemTap はもともと、**dprobes** や Linux Trace Toolkit などの以前の Linux プローブツールと同様の機能を Red Hat Enterprise Linux に提供するために開発されました。SystemTap は、カーネルアクティビティーを追跡するためのインフラストラクチャーをユーザーに提供することで、既存の Linux 監視ツールのスイートを補完することを目的としています。さらに、SystemTap はこの機能を 2 つの属性と組み合わせます。

- 柔軟性 - SystemTap のフレームワークを使用すると、幅広いカーネル機能、システムコール、およびカーネルスペースで発生する他のイベントについて、調査および監視目的のシンプルなスクリプトを開発できます。つまり、SystemTap は **ツール** というよりも、独自のカーネル固有のフォレンジックおよび監視ツールの開発を可能にするシステムといえます。
- 容易な使用 - 上記で説明したように、SystemTap を使うことでユーザーはカーネルプロセスのインストールメント化、再コンパイル、インストール、および再起動という面倒なプロセスを経ずにカーネルスペースのイベントをプローブできるようになります。

[4章 便利な SystemTap スクリプト](#) で挙げられている SystemTap スクリプトのほとんどは、他の同様の

ツール (**top**、**OProfile**、または **ps** など) ではネイティブで利用可能ではないシステムフォレンジックや監視機能の例になります。これらのスクリプトは、SystemTap のアプリケーションの豊富な例を示すために提供されており、これによりユーザーは、独自の SystemTap スクリプト作成時に使用可能な機能について学ぶことができます。

第2章 SYSTEMTAP の使用

本章では、SystemTap のインストール方法と SystemTap スクリプトの実行方法を説明します。

2.1. インストールと設定

SystemTap をデプロイするには、SystemTap パッケージと、対応するカーネルの `-devel`、`-debuginfo`、および `-debuginfo-common-arch` パッケージのセットをインストールする必要があります。システムに複数のカーネルがインストールされていて、それらのカーネル上で SystemTap を使用するには、それらの **each** カーネルバージョン用に `-devel` と `-debuginfo` パッケージをインストールします。

これらの手順は、以下のセクションで詳細に説明します。



重要

多くのユーザーは、`-debuginfo` と `-debug` パッケージを混同しています。SystemTap をデプロイするには、カーネルの `-debug` バージョンではなく、カーネルの `-debuginfo` パッケージをインストールする必要があることに注意してください。

2.1.1. SystemTap のインストール

SystemTap をデプロイするには、**root** として次のコマンドを実行して、`systemtap` および `systemtap-runtime` パッケージをインストールします。

```
~]# yum install -y systemtap systemtap-runtime
```

2.1.2. 必要なカーネル情報パッケージのインストール

SystemTap は、カーネル内にインストールメンテーションを配置する (プローブする) ためにカーネルの情報が必要になります。SystemTap がインストールメンテーション用のコードを生成できるようにするこの情報は、対応する `kernel-devel`、`kernel-debuginfo`、および `kernel-debuginfo-common-arch` パッケージに含まれています (ここで、`arch` はシステムのハードウェアプラットフォームです。**uname -m** コマンドを実行して判別します)。

`kernel-devel` パッケージはデフォルトの Red Hat Enterprise Linux リポジトリから入手できますが、`kernel-debuginfo` および `kernel-debuginfo-common-arch` パッケージは **debug** リポジトリから入手できます。

必要なパッケージをインストールするには、システムの **debug** リポジトリを有効にします。

```
~]# subscription-manager repos --enable=rhel-7-variant-debug-rpms
```

上記のコマンドで、使用している Red Hat Enterprise Linux システムのバリエーションに応じて、`variant` を **server**、**workstation**、または **client** に置き換えます。バリエーションを特定するには、次のコマンドを使用できます。

```
~]# cat /etc/redhat-release
Red Hat Enterprise Linux Server release 7.2 (Maipo)
```

`kernel-devel`、`kernel-debuginfo`、および `kernel-debuginfo-common-arch` パッケージのバージョン、バリエーション、およびアーキテクチャーは、SystemTap でプローブされるカーネルと **正確に** 一致する必要があります。システムが現在実行しているカーネルを特定するには、次を使用します。

```
uname -r
3.10.0-327.el7.x86_64
```

たとえば、AMD64 または Intel 64 マシンのカーネルバージョン **3.10.0-327.4.4.el7** で SystemTap を使用する場合は、次のパッケージをインストールする必要があります。

- kernel-debuginfo-3.10.0-327.4.4.el7.x86_64.rpm
- kernel-debuginfo-common-x86_64-3.10.0-327.4.4.el7.x86_64.rpm
- kernel-devel-3.10.0-327.4.4.el7.x86_64.rpm

yum パッケージマネージャーを使用して、現在のカーネルで SystemTap を実行するために必要なパッケージをインストールするには、**root** として次のコマンドを実行します。

```
~]# yum install -y kernel-devel-$(uname -r) \
kernel-debuginfo-$(uname -r) \
kernel-debuginfo-common-$(uname -m)-$(uname -r)
```

2.1.3. 初期テスト

SystemTap でプローブするカーネルが現在使用中であれば、デプロイメントが成功したかどうかを直ちにテストできます。別のカーネルをプローブする場合は、再起動して該当カーネルを読み込みます。

テストを開始するには、次のコマンドを実行します。

```
stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'
```

このコマンドは、**read performed** を出力し、仮想ファイルシステムの読み取りが検出されたら、適切に終了するように、SystemTap に指示するだけです。SystemTap が正常にデプロイされていれば、以下のような出力になります。

```
Pass 1: parsed user script and 45 library script(s) in 340usr/0sys/358real ms.
Pass 2: analyzed script: 1 probe(s), 1 function(s), 0 embed(s), 0 global(s) in 290usr/260sys/568real ms.
Pass 3: translated to C into "/tmp/stapiArgLX/stap_e5886fa50499994e6a87aacdc43cd392_399.c" in 490usr/430sys/938real ms.
Pass 4: compiled C into "stap_e5886fa50499994e6a87aacdc43cd392_399.ko" in 3310usr/430sys/3714real ms.
Pass 5: starting run.
read performed
Pass 5: run completed in 10usr/40sys/73real ms.
```

(**Pass 5** で始まる) 出力の最後の 3 行は、SystemTap がカーネルをプローブするインストールメンテーションを正常に作成できたこと、インストールメンテーションを実行したこと、プローブしているイベントを検出したこと (このケースでは、仮想ファイルシステムの読み込み)、および有効なハンドラーを実行したこと (テキストを出力し、エラーなしで終了) を示しています。

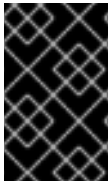
2.2. 他のコンピューター用のインストールメンテーション生成

ユーザーが SystemTap スクリプトを実行すると、そのスクリプトからカーネルモジュールが構築されます。すると SystemTap はそのモジュールをカーネルに読み込み、カーネルから直接指定されたデータを抽出できるようにします (詳細は、「[アーキテクチャー](#)」の [手順3.1「SystemTap セッション](#)」を参照)。

通常、SystemTap スクリプトは (「インストールと設定」にあるように) SystemTap がデプロイされているシステムでのみ実行できます。つまり、SystemTap を 10 台のシステムで実行するには、これらすべてのシステムに SystemTap をデプロイする必要があります。場合によっては、これは実現不可能もしくは望ましくないこともあります。たとえば、企業のポリシーで管理者が特定マシンにコンパイラやデバッグ情報を提供するパッケージのインストールを禁止されていれば、SystemTap のデプロイはできなくなります。

この状況を避けるためには、クロスインストールメンテーションを使用します。これは、1 台のコンピュータ上の SystemTap スクリプトから別のコンピュータで使用する SystemTap インストールメンテーションモジュールを生成するプロセスです。このプロセスは、以下の利点をもたらします。

- 各種マシンのカーネル情報パッケージを単一のホストマシンにインストールできます。
- 生成された SystemTap インストールメンテーションモジュール (systemtap-runtime) を使用するには、ターゲットマシンごとインストールする必要があるパッケージは 1 つだけです。



重要

構築されたインストールメンテーションモジュールが機能するには、ホストシステムとターゲットシステムが同一アーキテクチャーで同じ Linux ディストリビューションを実行している必要があります。



注記

本セクションでは分かりやすくするために、以下の用語を使用します。

インストールメンテーションモジュール

SystemTap スクリプトから構築したカーネルモジュール。SystemTap モジュールはホストシステム上に構築され、ターゲットシステムのターゲットカーネルに読み込まれます。

ホストシステム

ターゲットシステムに読み込めるように (SystemTap スクリプトから) インストールメンテーションモジュールをコンパイルするシステム。

ターゲットシステム

(SystemTap スクリプトから) インストールメンテーションモジュールを構築するシステム。

ターゲットカーネル

ターゲットシステムのカーネル。このカーネルがインストールメンテーションモジュールの読み込み、実行を行います。

手順2.1 ホストシステムとターゲットシステムの設定

1. 各ターゲットシステムに **systemtap-runtime** パッケージをインストールします。
2. 各ターゲットシステムで **uname -r** を実行して、各ターゲットシステムで実行中のカーネルを確認します。
3. SystemTap をホストシステムにインストールします。インストールメンテーションモジュールは、ホストシステム上でターゲットシステム用に構築されます。SystemTap のインストール方法については、「SystemTap のインストール」を参照してください。
4. 上記で判明したターゲットカーネルのバージョンを使用して、ターゲットカーネルと関連パッケージを「必要なカーネル情報パッケージのインストール」に説明されている方法でホ

ストシステムにインストールします。複数のターゲットシステムで異なるターゲットカーネルを使用している場合は、ターゲットシステムで使用しているカーネルごとにこのステップを繰り返します。

手順2.1「[ホストシステムとターゲットシステムの設定](#)」が完了すると、(いずれのターゲットシステム用の) インストールメンテーションモジュールもホストシステムでの構築が可能になります。

インストールメンテーションモジュールを構築するには、ホストシステムで以下のコマンドを実行します (適切な値を指定してください)。

```
stap -r kernel_version script -m module_name -p4
```

ここで、*kernel_version* はターゲットカーネルのバージョン (ターゲットマシンでの `uname -r` の出力) を指し、*script* はインストールメンテーションモジュールに変換されるスクリプトを指し、*module_name* はインストールメンテーションモジュールの目的の名前です。

インストールメンテーションモジュールがコンパイルされたら、ターゲットシステムにコピーして、以下のコマンドを使用して読み込みます。

```
staprun module_name.ko
```

たとえば、**3.10.0-327.4.4.el7**ターゲットカーネル用の **simple.stp** という名前の SystemTap スクリプトから **simple.ko** インストールメンテーションモジュールを作成するには、次のコマンドを使用します。

```
stap -r 2.6.32-53.el6 -e 'probe vfs.read {exit()}' -m simple -p4
```

これにより、**simple.ko** という名前のモジュールが作成されます。インストールメンテーションモジュール **simple.ko** を使用するには、これをターゲットシステムにコピーして、(ターゲットシステム上で) 以下のコマンドを実行します。

```
staprun simple.ko
```

2.3. SYSTEMTAP スクリプトの実行

SystemTap スクリプトはコマンド **stap** で実行されます。**stap** は、標準入力またはファイルから SystemTap スクリプトを実行できます。

stap および **staprun** を実行するには、システムに対する昇格された特権が必要です。ただし、すべてのユーザーが SystemTap 実行のためだけに **root** アクセスを付与されるわけではありません。たとえば、権限のないユーザーが自身のマシン上で SystemTap インストールメンテーションを実行する必要がある場合もあります。

通常のユーザーが **root** アクセスなしで SystemTap を実行できるようにするには、ユーザーを以下の両方のユーザーグループに追加します。

stapdev

このグループのメンバーは **stap** を使用して SystemTap スクリプトを実行したり、**staprun** を使用して SystemTap インストールメンテーションモジュールを実行したりできます。

stap の実行では、SystemTap スクリプトがカーネルモジュールにコンパイルされ、それがカーネルに読み込まれます。これにはシステムに対する権限の昇格が必要となり、**stapdev** メンバーにはそれが付与されます。ただし、この権限は **stapdev** メンバーに有効な **root** アクセスも付与することになります。このため、**stapdev** グループのメンバーシップは、**root** アクセスを信頼して付与できるメンバーにのみ許可してください。

stapusr

このグループのメンバーが SystemTap インストールメンテーションモジュールの実行に使用できるのは、**staprun**のみです。さらに、これらのモジュールは `/lib/modules/kernel_version/systemtap/` からのみ実行できます。このディレクトリーを所有できるのは **root** ユーザーのみで、書き込みが可能なのも **root** ユーザーのみでなければならない点に注意してください。



SYSTEMTAP スクリプトの実行

SystemTap スクリプトを実行するには、ユーザーは `stapdev` と `stapusr` の **両方** のグループに属する必要があります。

以下は、一般的に使用される **stap** オプションのリストです。

-v

SystemTap セッションの出力をさらに詳細なものにします。このオプション (たとえば、**stap -vvv script.stp**) を繰り返して、スクリプトの実行に関する詳細を提供できます。スクリプトの実行中にエラーが発生すると、これは特に便利なものになります。

SystemTap スクリプトの一般的なエラーに関する詳細は、[5章 SystemTap のエラーを理解する](#) を参照してください。

-o filename

標準出力を *filename* に送信します。

-S size, count

ファイルのサイズを *size* で指定されたメガバイト数に制限し、ファイル数を *count* の数に制限します。ファイル名には、連続番号の接尾辞が付きます。このオプションは、SystemTap に `logrotate` 演算を実装します。

-o とともに使用すると、**-S** はログファイルのサイズを制限します。

-x process ID

SystemTap ハンドラー関数の **target()** を指定されたプロセス ID に設定します。**target()** の詳細は [SystemTap 関数](#) を参照してください。

-c command

SystemTap ハンドラー関数の **target()** を指定されたコマンドに設定します。指定コマンドへの完全パスを使用する必要があります。たとえば、**cp** を指定する代わりに、(**stap script -c/bin/cp** のように) `/bin/cp` を使用します。**target()** の詳細については、[SystemTap 関数](#) を参照してください。

-e 'script'

systemtap 翻訳の入力に、ファイルではなく **script** 文字列を使用します。

-F

SystemTap のフライトレコーダーモードを使用し、スクリプトをバックグラウンドプロセスにします。フライトレコーダーモードの詳細は、「[SystemTap フライトレコーダーモード](#)」を参照してください。

stap は、`-` スイッチを使用して標準入力からスクリプトを実行するように指示することもできます。例を示します。

例2.1 標準入力からスクリプトを実行

```
echo "probe timer.s(1) {exit()}" | stap -
```

例2.1「標準入力からスクリプトを実行」は、**stap** に、**echo** によって標準入力に渡されたスクリプトを実行するように指示します。**stap** で使用するオプションはすべて、`-` スイッチの前に挿入する必要があります。たとえば、例2.1「標準入力からスクリプトを実行」をより詳細な出力にするには、以下のようなコマンドになります。

```
echo "probe timer.s(1) {exit()}" | stap -v -
```

stap の詳細については、**man stap** を参照してください。

SystemTap インストールメンターション (クロスインストールメンターション中に SystemTap スクリプトから構築されたカーネルモジュール) を実行するには、**staprun** を代わりに使用します。**staprun** とクロスインストールメンターションの詳細については、「[他のコンピューター用のインストールメンターション生成](#)」を参照してください。



注記

stap オプション `-v` および `-o` も **staprun** で機能します。**staprun** の詳細については、`staprun(1) man` ページを参照してください。

2.3.1. SystemTap フライトレコーダーモード

SystemTap のフライトレコーダーモードを使用すると、SystemTap スクリプトは長期間の実行が可能になり、最近の出力のみにフォーカスできるようになります。フライトレコーダーモード (`-F` オプション) は、生成される出力の量を制限します。フライトレコーダーモードには、メモリー内モードとファイルモードの2種類があります。どちらの場合も、SystemTap スクリプトはバックグラウンドプロセスとして実行されます。

2.3.1.1. メモリー内フライトレコーダー

フライトレコーダーモード (`-F` オプション) をファイル名なしで使用すると、SystemTap はカーネルメモリー内のバッファを使用してスクリプトの出力を保存します。次に、SystemTap インストールメンターションモジュールが読み込まれてプローブの実行が開始し、インストールメンターションが外されてバックグラウンドに置かれます。関心のあるイベントが発生すると、インストールメンターションは再度アタッチされ、メモリーバッファ内の最近の出力と継続中の出力が閲覧可能となります。以下のコマンドでは、フライトレコーダーのメモリー内モードを使用してスクリプトが開始されます。

```
stap -F /usr/share/doc/systemtap-version/examples/io/iotime.stp
```

スクリプトが開始されると、実行中のスクリプトに再接続するためのコマンドを示すメッセージが表示されます。

```
Disconnecting from systemtap module. To reconnect, type "staprun -A
stap_5dd0073edcb1f13f7565d8c343063e68_19556"
```

関心のあるイベントが発生したら、実行中のスクリプトに再度アタッチしてメモリーバッファ内の最近のデータを出力し、継続中の出力を得るために、以下のコマンドを使用します。

```
staprun -A stap_5dd0073edcb1f13f7565d8c343063e68_19556
```

カーネルバッファはデフォルトでは 1MB のサイズですが、**-s** オプションを使用してバッファのメガバイト単位のサイズを指定する (2 の累乗に切り上げる) ことが可能です。たとえば、SystemTap コマンドライン上で **-s2** とすると、バッファを 2MB に指定します。

2.3.1.2. ファイルフライトレコーダー

フライトレコーダーモードは、ファイルにデータを保存することもできます。保持するファイルの数とサイズは、**-S** オプションにコンマ区切りの 2 つの数字の引数を続けて制御します。最初の引数は、各出力ファイルのメガバイト単位の最大サイズです。2 つ目の引数は、保存する最新ファイルの数です。ファイル名は、**-o** オプションの後に名前を付けて指定します。SystemTap は数値の接尾辞をファイル名に付けてファイルの順序を示します。以下のコマンドでは、SystemTap がフライトレコーダーモードで開始され、出力は **/tmp/pfaults.log.[0-9]+** という名前のファイルに保存されます。各ファイルのサイズは 1MB 以下となり、最新の 2 ファイルが保存されます。

```
stap -F -o /tmp/pfaults.log -S 1,2 pfaults.stp
```

コマンドが出力する数字は、プロセス ID です。プロセスに **SIGTERM** を送信すると、SystemTap スクリプトはシャットダウンし、データ収集が停止されます。たとえば、以前のコマンドでプロセス ID が **7590** となっている場合、以下のコマンドを実行すると SystemTap スクリプトはシャットダウンします。

```
kill -s SIGTERM 7590
```

スクリプトが生成した最新の 2 ファイルのみが保存され、それより古いファイルは削除されます。したがって、**ls -sh/tmp/pfaults.log.*** は次の 2 つのファイルのみを示します。

```
1020K /tmp/pfaults.log.5 44K /tmp/pfaults.log.6
```

最新のデータは数字の大きい方のファイルで、上記の場合は **/tmp/pfaults.log.6** になります。

第3章 SYSTEMTAP の作動方法

SystemTap を使うとユーザーはシンプルなスクリプトを作成および再使用して、実行中の Linux システムのアクティビティーを徹底的に調べることができます。このスクリプトは、データを抽出し、フィルターをかけ、素早く安全にデータをまとめるように設計できます。これにより、複雑なパフォーマンス (または機能的な) 問題の診断が可能になります。

SystemTap スクリプトにおける本質的な考えは、イベントに名前を付け、それにハンドラーを与えることです。SystemTap がスクリプトを実行すると、SystemTap はイベントを監視します。イベントが発生すると、Linux カーネルはハンドラーをクイックサブルーチンとして実行し、通常の操作を再開します。

イベントには関数の開始/終了、タイマーの有効期限切れ、セッション終了など数種類のものがあります。ハンドラーは一連のスクリプト言語のステートメントで、イベント発生時に実行する作業を指定します。この作業には通常、イベントコンテキストからのデータ抽出、それらの内部変数への保存、結果の出力などが含まれます。

3.1. アーキテクチャー

SystemTap のセッションは、SystemTap スクリプトを実行すると始まります。このセッションは、以下の順番で生じます。

手順3.1 SystemTap セッション

1. まず、SystemTap はスクリプトを既存のタップセットライブラリー (通常は `/usr/share/systemtap/tapset/` ディレクトリーにあります) に対してチェックし、使用されているタップセットを探します。次に SystemTap は、見つかった tapset を tapset ライブラリー内の対応する定義で置き換えます。
2. SystemTap はスクリプトを C に変換し、システム C コンパイラーを実行してそこからカーネルモジュールを作成します。このステップを実行するツールは、systemtap パッケージに含まれています (詳細については、「[SystemTap のインストール](#)」を参照)。
3. SystemTap はモジュールを読み込み、スクリプト内の全プローブ (イベントおよびハンドラー) を有効にします。systemtap-runtime パッケージの `staprun` (詳細については、「[SystemTap のインストール](#)」を参照) がこの機能を提供します。
4. イベントが発生すると、それに対応するハンドラーが実行されます。
5. SystemTap セッションが終了すると、プローブは無効になり、カーネルモジュールは読み込み解除されます。

このステップは、コマンドラインプログラム `stap` で実行されます。このプログラムは、SystemTap のメインのフロントエンドツールです。`stap` の詳細については、`stap(1) man` ページを参照してください (マシンに SystemTap が適切にインストールされている場合)。

3.2. SYSTEMTAP スクリプト

SystemTap スクリプトはそのほとんどにおいて、各 SystemTap セッションのベースになっています。SystemTap スクリプトが SystemTap に対してどのタイプの情報を収集するか、収集後に何をするかを指示します。

[3章 SystemTap の作動方法](#) の説明にあるように、SystemTap スクリプトは **イベント** と **ハンドラー** という2つのコンポーネントで設定されています。SystemTap セッションが開始されたら、SystemTap はオペレーティングシステムで指定されたイベントを監視し、イベントが発生したらハンドラーを実行

します。



注記

イベントとそれに対応するハンドラーは、まとめて **プローブ** と呼ばれます。SystemTap スクリプトには複数のプローブを備えることができます。プローブのハンドラーは一般的に **プローブボディー** と呼ばれます。

アプリケーションの開発という面では、イベントとハンドラーの使用は診断出力ステートメントをコマンドのプログラムシーケンスに挿入するというコードのインストルメント化に似ています。診断出力ステートメントを使用すると、プログラムの実行後に発行されたコマンドの履歴をみることができます。

SystemTap スクリプトでは、コードを再コンパイルすることなくインストルメンテーションコードの挿入が可能で、ハンドラーに関する柔軟性が広がります。イベントは、ハンドラー実行の引き金となります。ハンドラーは指定されたデータを記録して、特定の方法で出力するよう指定できます。

形式

SystemTap スクリプトは、ファイル拡張子 **.stp** を使用し、以下の書式のプローブを含んでいます。

```
probe event {statements}
```

SystemTap は1つのプローブにつき複数のイベントをサポートしており、複数イベントはコンマ(,)で区切ります。1つのプローブで複数のイベントが指定された場合、SystemTap は指定されたイベントが発生するとそのハンドラーを実行します。

プローブにはそれぞれ、対応する **ステートメントブロック** があります。このステートメントブロックは中括弧({})で囲まれており、イベントごとに実行されるステートメントが含まれています。SystemTap はこれらのステートメントを順番に実行し、通常は複数のステートメントを分ける特別なセパレーターやターミネーターは必要ありません。



注記

SystemTap スクリプト内のステートメントブロックは、C プログラミング言語と同じ構文とセマンティクスを使用します。ステートメントブロックは、別のステートメントブロック内の入れ子状態にすることができます。

Systemtap では、多くのプローブが使用するコードを外に括り出して関数を作成することができます。つまり、複数のプローブで同じステートメントを何度も繰り返し書くのではなく、以下のように *function* 内に指示を配置することができます。

```
function function_name(arguments){statements}
probe event {function_name(arguments)}
```

function_name の *statements* は、*event* のプローブが実行されるときに実行されます。*arguments* は、*function* に渡されるオプションの値です。



重要

「[SystemTap スクリプト](#)」では、SystemTap スクリプトの基本を説明します。SystemTap スクリプトの理解を深めるには、[4章 便利な SystemTap スクリプト](#) を参照してください。この章の各セクションでは、スクリプト、イベント、ハンドラー、および予測される出力を詳細に説明しています。

3.2.1. イベント

SystemTap のイベントは大まかに、*同期*と *非同期*に分けられます。

同期イベント

同期 イベントは、プロセスがカーネルコード内の特定の場所で指示を実行する際に発生します。これは他のイベントの参照ポイントとなり、ここからさらにコンテキストデータが入手可能になります。

同期イベントの例には以下のようなものがあります。

`syscall.system_call`

システムコール `system_call` へのエントリー。システムコールの終了を希望する場合は、`.return` をイベントに追加すると、システムコールの終了を監視ようになります。たとえば、システムコール `close` のエントリーと終了を指定するには、`syscall.close` および `syscall.close.return` をそれぞれ使用します。

`vfs.file_operation`

仮想ファイルシステム (VFS) の `file_operation` イベントへのエントリー。`syscall` イベントと同様に、イベントに `.return` を追加すると、`file_operation` 動作の終了を監視します。

`kernel.function("function")`

カーネル関数 `function` へのエントリー。たとえば、`kernel.function("sys_open")` は、カーネル関数 `sys_open` がシステム内のスレッドによって呼び出される際に発生する **イベント** を参照します。カーネル関数 `sys_open` の `return` を指定するには、`kernel.function("sys_open").return` のように、`return` 文字列をイベントステートメントに追加します。

プローブイベントを定義する際には、アスタリスク (*) をワイルドカードに使用できます。また、カーネルソースファイル内の関数のエントリーまたは終了も追跡可能です。以下の例を見てみましょう。

例3.1 wildcards.stp

```
probe kernel.function("**@net/socket.c") {}
probe kernel.function("**@net/socket.c").return {}
```

この例では、最初のプローブのイベントは `net/socket.c` カーネルソースファイル内の全関数のエントリーを指定しています。2つ目のプローブでは、これら全関数の終了を指定しています。この例では、ハンドラーにステートメントがないことに注意してください。このため、情報が収集されず、表示されることもありません。

`kernel.trace("tracepoint")`

`tracepoint` の静的プローブ。最近のカーネル (2.6.30 以降) には、カーネル内の特定イベント用のインストルメンテーションが含まれています。これらのイベントは、トレースポイントで静的にマークが付けられています。SystemTap で利用可能なトレースポイントの例としては、`kernel.trace("kfree_skb")` があります。これは、カーネル内でネットワークバッファが解放されると合図します。

`module("module").function("function")`

モジュール内の関数のプローブを可能にします。以下に例を示します。

例3.2 moduleprobe.stp

```
probe module("ext3").function("*") { }
probe module("ext3").function("*").return { }
```

例3.2 「[moduleprobe.stp](#)」の最初のプローブは、**ext3** モジュールの全関数のエントリーを指しています。2つ目のプローブは、同じモジュールの全関数の終了を指しています。**.return** の接尾辞の使用は、**kernel.function()** と類似します。例3.2 「[moduleprobe.stp](#)」のプローブは、プローブハンドラーにステートメントがないことに注意してください。このため、有用なデータは出力されません(例3.1 「[wildcards.stp](#)」の場合と同様)。

システムのカーネルモジュールは通常、`/lib/modules/kernel_version` にあります。ここで、`kernel_version` は現在ロードされているカーネルバージョンを指します。モジュールは、ファイル名拡張子 **.ko** を使用します。

非同期イベント

非同期イベントは、コード内の特定の指示や場所に関連付けられていません。このタイプのプローブイベントは、主にカウンターやタイマー、同様のコンストラクトで設定されています。

非同期イベントの例には以下のようなものがあります。

begin

SystemTap スクリプトの実行と同時に、SystemTap セッションが起動します。

end

SystemTap セッションが終了します。

timer イベント

ハンドラーの定期実行を指定するイベントです。以下に例を示します。

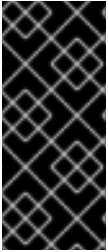
例3.3 timer-s.stp

```
probe timer.s(4)
{
  printf("hello world\n")
}
```

例3.3 「[timer-s.stp](#)」は、4秒ごとに **hello world** を出力するプローブの例です。以下のような timer イベントも使用できます。

- **timer.ms**(*milliseconds*)
- **timer.us**(*microseconds*)
- **timer.ns**(*nanoseconds*)
- **timer.hz**(*hertz*)
- **timer.jiffies**(*jiffies*)

情報を収集する他のプローブと併せて timer イベントを使用すると、定期的な更新を出力でき、その情報の変遷が分かります。



重要

SystemTap では、多数のプロブイベントの使用をサポートしています。サポートされているイベントの詳細については、`stapprobes(3) man` ページを参照してください。stapprobes(3) の『SEE ALSO』セクションには、特定のサブシステムやコンポーネントでサポートされているイベントについて説明している他の man ページへのリンクも含まれています。

3.2.2. Systemtap ハンドラー/ボディー

以下のサンプルスクリプトを見てみましょう。

例3.4 helloworld.stp

```
probe begin
{
    printf ("hello world\n")
    exit ()
}
```

例3.4 「helloworld.stp」では、**begin** イベント (セッションの開始) が { } で囲まれているハンドラーを始動させます。これは単に **hello world** を出力して改行し、終了するものです。



注記

SystemTap スクリプトの実行は、**exit()** 関数が実行されるまで継続されます。スクリプトの実行を停止したい場合は、手動で **Ctrl+C** と入力すると中断できます。

printf () ステートメント

printf() ステートメントは、データを出力するための最も単純な関数の一つです。**printf()** は、次の形式で多くの SystemTap 関数を使用してデータを表示するためにも使用できます。

```
printf ("format string\n", arguments)
```

format string では、*arguments* の出力方法を指定します。例3.4 「helloworld.stp」の書式文字列は、SystemTap に **hello world** の出力を指示するだけで、書式指定子は含みません。

引数によっては、**%s** (文字列用) や **%d** (数字用) といった書式指定子を書式文字列に使用することもできます。書式文字列には複数の書式指定子を使用することが可能で、それぞれを対応する引数に一致させます。複数の引数はコンマ (,) で区切ります。



注記

セマンティックの面では、SystemTap **printf** 関数は、C 言語の関数に非常によく似ています。上記の SystemTap の **printf** 関数における構文と書式は、C 言語の **printf** と同一のものであります。

以下のプロブの例を見てみましょう。

例3.5 variables-in-printf-statements.stp

```
probe syscall.open
{
    printf ("%s(%d) open\n", execname(), pid())
}
```

例3.5 「[variables-in-printf-statements.stp](#)」では、SystemTap がシステムコール **open** への全エントリをプローブするように指示しています。各イベントでは、現行の **execname()** (実行可能ファイル名の付いた文字列) と **pid()** (現行のプロセス ID 番号) に続けて **open** という単語を出力します。このプローブ出力の抜粋は以下のようになります。

```
vmware-guestd(2206) open
hald(2360) open
hald(2360) open
hald(2360) open
df(3433) open
df(3433) open
df(3433) open
hald(2360) open
```

SystemTap 関数

SystemTap は、**printf()** 引数として使用できる多くの機能をサポートしています。例3.5 「[variables-in-printf-statements.stp](#)」は、SystemTap 関数 **execname()** (カーネル関数を呼び出した/システムコールを実行したプロセスの名前) および **pid()** (現在のプロセス ID) を使用します。

一般的に使用される SystemTap 関数を以下に挙げます。

tid()

現行スレッドの ID。

uid()

現行ユーザーの ID。

cpu()

現行の CPU 番号。

gettimeofday_s()

Unix epoch (1970 年 1 月 1 日) からの秒数。

ctime()

UNIX epoch からの秒数を日にちに換算。

pp()

現在処理されているプローブポイントを記述する文字列。

thread_indent()

この関数は出力結果をうまく整理するので、便利なものです。この関数はインデント差分の引数を取ります。これは、スレッドのインデントカウンターに追加する、またはそこから取り除くスペースの数を示すものです。その後、適切なインデントスペースの数と一般的な追跡データの文字列を返します。

ここで返された文字列の一般的なデータに含まれるのは、タイムスタンプ (スレッドの `thread_indent()` への最初の呼び出しからのマイクロ秒)、プロセス名、およびスレッド ID です。これによりどの関数が呼び出されたか、誰が呼び出したか、各関数呼び出しの長さが特定できます。

それぞれの呼び出しが終わり次第、次の呼び出しが始まれば、エントリーと終了の一致は容易です。ただし、ほとんどの場合は、最初の関数呼び出しエントリーがなされた後、これが終了する前に他の複数の呼び出しがエントリーおよび終了することがあります。インデントカウンターがあることで、最初の呼び出しが終了していない場合、次の関数呼び出しをインデントして、エントリーとそれに対応する終了が一致しやすくなります。

以下で `thread_indent()` の使用例を見てみましょう。

例3.6 thread_indent.stp

```
probe kernel.function("@net/socket.c")
{
    printf ("%s -> %s\n", thread_indent(1), probefunc())
}
probe kernel.function("@net/socket.c").return
{
    printf ("%s <- %s\n", thread_indent(-1), probefunc())
}
```

例3.6 「`thread_indent.stp`」では、各イベントでの `thread_indent()` と `probe` の関数を以下の書式で出力します。

```
0 ftp(7223): -> sys_socketcall
1159 ftp(7223): -> sys_socket
2173 ftp(7223): -> __sock_create
2286 ftp(7223): -> sock_alloc_inode
2737 ftp(7223): <- sock_alloc_inode
3349 ftp(7223): -> sock_alloc
3389 ftp(7223): <- sock_alloc
3417 ftp(7223): <- __sock_create
4117 ftp(7223): -> sock_create
4160 ftp(7223): <- sock_create
4301 ftp(7223): -> sock_map_fd
4644 ftp(7223): -> sock_map_file
4699 ftp(7223): <- sock_map_file
4715 ftp(7223): <- sock_map_fd
4732 ftp(7223): <- sys_socket
4775 ftp(7223): <- sys_socketcall
```

この出力サンプルには、以下の情報が含まれています。

- スレッドの最初の `thread_indent()` 呼び出しからの時間 (マイクロ秒単位)。
- 関数呼び出しを行ったプロセス名 (およびそれに対応する ID)。
- 呼び出しがエントリー (<-) か終了 (->) かを示す矢印。インデントがあることで、特定の関数呼び出しのエントリーとそれに対応する終了が一致しやすくなります。
- プロセスが呼び出した関数名。

name

特定のシステムコールの名前を識別します。この変数は、イベント `syscall.system_call` を使用するプローブでのみ使用できます。

target()

次の2つのコマンドのいずれかと組み合わせて使用します。

```
stap script -x process ID stap script -c command
```

プロセス ID またはコマンドの引数を取るスクリプトを指定したい場合、スクリプト内で参照先となる変数として `target()` を使用します。以下に例を示します。

例3.7 targetexample.stp

```
probe syscall.* {
  if (pid() == target())
    printf("%s/n", name)
}
```

例3.7「[targetexample.stp](#)」を引数 `-x process ID` で実行すると、(`syscall.*` イベントで指定された)すべてのシステムコールを監視し、指定されたプロセスで実行された全システムコールの名前を出力します。

これは、特定のプロセスをターゲットとしたい場合に毎回 `if (pid() == process ID)` と指定することと同様の効果があります。ただし、`target()` を使用すると、スクリプトの再利用が容易になり、スクリプトを実行するたびに引数としてプロセス ID を渡すだけで済みます。以下に例を示します。

```
stap targetexample.stp -x process ID
```

サポートされる SystemTap 関数の詳細情報は、`stapfuncs(3)` を参照してください。

3.3. 基本的な SYSTEMTAP ハンドラーコンストラクト

SystemTap は、ハンドラーでいくつかの基本的なコンストラクトの使用をサポートしています。ほとんどのこれらハンドラーコンストラクトの構文は、C および `awk` 構文に基づいています。本セクションでは、SystemTap ハンドラーコンストラクトで最も有用なものをいくつか説明します。これで、簡潔かつ便利な SystemTap スクリプトの作成ができるようになります。

3.3.1. 変数

ハンドラーでは、変数を自由に使うことができます。名前を選択し、関数もしくは式から値を割り当て、式内で値を使用します。SystemTap は、割り当てられた値のタイプに基づいて、自動的に変数が文字列か整数かを識別します。たとえば、変数 `var` を `gettimeofday_s()` に (`var = gettimeofday_s()` のように) 設定すると、`var` は数字として識別され、`printf()` では整数の書式指定子 (`%d`) で出力されます。

ただしデフォルトでは、変数は使用されているプローブのみのローカルとなります。つまり、変数はプローブハンドラーが呼び出されるたびに初期化され、使用され、破棄されます。複数のプローブで変数を共有するには、プローブの外で `global` を使用して変数名を宣言します。以下の例を見てみましょう。

■

例3.8 timer-jiffies.stp

```

global count_jiffies, count_ms
probe timer.jiffies(100) { count_jiffies ++ }
probe timer.ms(100) { count_ms ++ }
probe timer.ms(12345)
{
  hz=(1000*count_jiffies) / count_ms
  printf ("jiffies:ms ratio %d:%d => CONFIG_HZ=%d\n",
    count_jiffies, count_ms, hz)
  exit ()
}

```

例3.8 「timer-jiffies.stp」では、jiffies およびミリ秒をカウントするタイマーを使用してカーネルの **CONFIG_HZ** 設定を計算し、それに応じて計算しています。global ステートメントにより、スクリプトは変数 **count_jiffies** および **count_ms** (各プローブで設定) を **probe timer.ms(12345)** と共有して使用することができます。

注記

例3.8 「timer-jiffies.stp」の ++ 表記 (**count_jiffies ++** と **count_ms ++**) は、変数の値を1増やすために使用されます。以下のプローブで **count_jiffies** は、100 jiffies ごとに1つ増えます。

```

probe timer.jiffies(100) { count_jiffies ++ }

```

この場合、SystemTap は **count_jiffies** が整数であると理解します。**count_jiffies** には初期値が割り当てられていないことから、デフォルトでは初期値はゼロになります。

3.3.2. 条件付き (conditional) ステートメント

場合によっては、SystemTap スクリプトの出力が大きすぎることもあるかもしれません。これに対処するには、スクリプトの論理を細分化して、出力をプローブに関連するもしくは有用なものにする必要があります。

これはハンドラーで **条件** を使うことで実行できます。SystemTap は以下のタイプの条件付きステートメントを受け付けます。

If/Else ステートメント

書式は以下のようになります。

```

if (condition)
  statement1
else
  statement2

```

condition 式がゼロ以外の場合、**statement1** が実行されます。**condition** 式がゼロの場合、**statement2** が実行されます。**else** 節 (**else statement2**) はオプションです。**statement1** と **statement2** は両方とも、ステートメントブロックになります。

例3.9 ifelse.stp

```

global countread, countnonread
probe kernel.function("vfs_read"),kernel.function("vfs_write")
{
  if (probefunc()=="vfs_read")
    countread ++
  else
    countnonread ++
}
probe timer.s(5) { exit() }
probe end
{
  printf("VFS reads total %d\n VFS writes total %d\n", countread, countnonread)
}

```

例3.9 「ifelse.stp」は、5秒間にシステムが実行する仮想ファイルシステムの読み込み (**vfs_read**) と書き込み (**vfs_write**) 回数をカウントするスクリプトです。この実行時には、プローブした関数の名前が (条件 **if (probefunc()=="vfs_read")** に記してあるように) **vfs_read** と一致する場合、スクリプトは変数 **countread** の値を1つ増やします。一致しない場合は、**countnonread** (**else {countnonread ++}**) を増やします。

While ループ

書式は以下のようになります。

```

while (condition)
  statement

```

condition がゼロ以外であれば、**statement** 内のステートメントのブロックは実行されます。**statement** はステートメントブロックであることが多く、**condition** が最終的にゼロとなるように値を変更する必要があります。

For ループ

書式は以下のようになります。

```

for (initialization; conditional; increment) statement

```

for ループは、単に **while** ループの短縮形です。以下が同等の **while** ループになります。

```

initialization
while (conditional) {
  statement
  increment
}

```

条件演算子

等号 **==** のほかに、以下の演算子も条件付きステートメントに使用できます。

>=

より大か等しい

<=

より小か等しい

!=

等しくない

3.3.3. コマンドラインの引数

SystemTap スクリプトでは、**\$** または **@** の直後にコマンドライン上の引数の番号を続けることで、単純なコマンドライン引数を受け付けるようにすることができます。コマンドライン引数としてユーザーが整数を入力すると思われる場合は **\$** を、文字列が予測される場合は **@** を使用します。

例3.10 commandlineargs.stp

```
probe kernel.function(@1) { }
probe kernel.function(@1).return { }
```

例3.10 「[commandlineargs.stp](#)」は例3.1 「[wildcards.stp](#)」と似ていますが、(**stap commandlineargs.stp kernel function** のように) プローブするカーネル関数をコマンドライン引数として渡すことができる点で異なります。また、**@1**、**@2** のように、ユーザーが入力した順番で複数のコマンドライン引数をスクリプトが受け付けるように指定することもできます。

3.4. 連想アレイ

SystemTap は、連想アレイの使用もサポートします。通常の変数は単一の値を表しますが、連想アレイでは値の集合を表すことができます。簡単に言うと、連想アレイは一意的キーの集合です。アレイ内のそれぞれのキーにそれに関連する値があります。

(これより後で紹介するように) 連想アレイは通常複数のプローブで処理されるので、SystemTap スクリプトでは **global** 変数として宣言されるべきです。連想アレイ内の要素にアクセスする構文は **awk** の構文と似ており、以下のようになります。

```
array_name[index_expression]
```

ここで **array_name** は、アレイが使用する任意の名前です。**index_expression** は、アレイ内の特定の一意のキーを見るために使用されます。例として、**tom**、**dick**、および **harry** という3人の年齢(一意のキー)を指定する、**arr** という名前のアレイを構築してみましょう。それらにそれぞれ23、24、および25の年齢(関連付けられた値)を割り当てるには、次のアレイステートメントを使用します。

例3.11 基本的なアレイステートメント

```
arr["tom"] = 23
arr["dick"] = 24
arr["harry"] = 25
```

アレイステートメント内では最大9つのインデックス式を指定することができ、それぞれをコンマ(,)で区切ります。これは、キーに複数の情報が含まれる場合に便利です。例4.9 「[disktop.stp](#)」からの以下の行では、プロセスID、実行可能ファイル名、ユーザーID、親プロセスのID、および文字列Wという5つの要素を使用しています。ここでは、**devname** の値をキーに関連付けています。

```
device[pid(),execname(),uid(),ppid(),"W"] = devname
```



重要

連想アレイは、単一プローブで使用されるか複数プローブで使用されるかに関わらず、すべてを **global** と宣言する必要があります。

3.5. SYSTEMTAP でのアレイ演算

本セクションでは、SystemTap で使用される最も一般的なアレイ演算を説明しています。

3.5.1. 関連する値の割り当て

インデックス化された一意のペアに関連する値を設定するには、等号 `=` を使います。

```
array_name[index_expression] = value
```

例3.11「基本的なアレイステートメント」では、明示的な関連する値を一意のキーに設定する非常に基本的な例を示しています。**index_expression** と **value** の両方に、ハンドラー関数を使用することもできます。たとえば以下のように、アレイを使用して、タイムスタンプをプロセス名(これを一意のキーとして使用)への関連する値として設定することができます。

例3.12 タイムスタンプをプロセス名に関連付ける

```
arr[tid()] = gettimeofday_s()
```

例3.12「タイムスタンプをプロセス名に関連付ける」ではイベントがステートメントを呼び出すと、SystemTap は適切な **tid()** 値(つまり、スレッドの ID。これは一意のキーとして使用されます)を返します。同時に SystemTap は関数 **gettimeofday_s()** を使用して、対応するタイムスタンプを関数 **tid()** で定義されている一意のキーへの関連する値として設定します。これで、スレッド ID とタイムスタンプを含むキーのペアで設定されるアレイが作成されます。

この例では、**tid()** がアレイ **arr** ですすでに定義されている値を返すと、この演算子はその値に関連付けられている元の値を破棄し、**gettimeofday_s()** からの現行タイムスタンプで置き換えます。

3.5.2. アレイからの値の読み取り

アレイからの値の読み取りは、変数値の読み取りと同じ方法でできます。これを行うには、**array_name[index_expression]** ステートメントを数式に要素として含めます。以下に例を示します。

例3.13 単純計算でのアレイ値の使用

```
delta = gettimeofday_s() - arr[tid()]
```

この例では、(「関連する値の割り当て」からの)例3.12「タイムスタンプをプロセス名に関連付ける」のコンストラクトを使用してアレイ **arr** が構築されていることを想定しています。これで参照ポイントとなるタイムスタンプが設定され、**delta** の計算に使用されます。

例3.13「単純計算でのアレイ値の使用」のコンストラクトは、現行の `gettimeofday_s()` からキー `tid()` の関連する値を差し引くことで、変数 `delta` の値を計算します。このコンストラクトは、`tid()` の値をアレイから読み取ることで計算を行います。このコンストラクトは、読み取り操作の開始と完了など、2つのイベント間の時間を判定する際に便利なものです。



注記

`index_expression` が一意のキーを見つけられない場合は、0の値 (例3.13「単純計算でのアレイ値の使用」などの数値演算の場合) もしくは `null` (空) の文字列の値 (文字列の演算の場合) がデフォルトで返されます。

3.5.3. 関連する値の増加

アレイ内の一意のキーの関連する値を増やすには、`++` を使用します。

```
array_name[index_expression] ++
```

ここでも、`index_expression` にハンドラー関数を使用できます。たとえば、仮想ファイルシステムへの読み込み (`vfs.read` イベントを使用) を特定のプロセスが実行した回数を計算したい場合は、以下のプローブを使用します。

例3.14 `vfsreads.stp`

```
probe vfs.read
{
  reads[execname()] ++
}
```

例3.14「`vfsreads.stp`」では、プローブが最初にプロセス名 `gnome-terminal` を返す際 (つまり、`gnome-terminal` が初めて VFS 読み込みを実行する際)、そのプロセス名は一意のキー `gnome-terminal` に関連する値1が付いたものになります。プローブがプロセス名 `gnome-terminal` を次に返す際には、SystemTap は `gnome-terminal` の関連する値を1増やします。SystemTap は、プローブがプロセス名を返す際にこの演算をすべてのプロセス名に対して実行します。

3.5.4. アレイ内での複数要素の処理

アレイで十分な情報を収集したら、それを有用なものにするためにアレイで全要素を取得して処理する必要があります。例3.14「`vfsreads.stp`」を見てみましょう。このスクリプトは各プロセスが何回 VFS 読み込みを行ったかという情報を収集しますが、その情報をどうするかについては指定していません。例3.14「`vfsreads.stp`」を有用にする簡単な方法は、`reads` アレイでキーのペアを出力することです。

アレイ内のキーのペアすべてを処理する最善の方法 (反復として) は、`foreach` ステートメントを使用することです。以下の例を見てみましょう。

例3.15 `cumulative-vfsreads.stp`

```
global reads
probe vfs.read
{
  reads[execname()] ++
}
```

```
probe timer.s(3)
{
  foreach (count in reads)
    printf("%s : %d \n", count, reads[count])
}
```

例3.15 「[cumulative-vfsreads.stp](#)」の2つ目のプローブでは、**foreach** ステートメントが **count** 変数を使用して **reads** アレイ内の一意のキーの反復を参照しています。同じプローブ内の **reads[count]** アレイステートメントは、一意のキーの関連する値を取得します。

例3.15 「[cumulative-vfsreads.stp](#)」の最初のプローブでは、スクリプトは VFS-read の統計情報を 3 秒ごとに出だし、VFS-read を実行したプロセス名とその回数を表示します。

例3.15 「[cumulative-vfsreads.stp](#)」の **foreach** ステートメントは、順不同でアレイ内のプロセス名の全反復を出力することに注意してください。**+** (昇順) または **-** (降順) を使用すると、スクリプトに特定の順番で反復をプロセスするよう指示することができます。さらに、**limit value** オプションを使うと、スクリプトがプロセスする反復数を制限することもできます。

以下の代替プローブ例を見てみましょう。

```
probe timer.s(3)
{
  foreach (count in reads- limit 10)
    printf("%s : %d \n", count, reads[count])
}
```

この **foreach** ステートメントは、スクリプトにアレイ **reads** 内の要素を (関連する値の) 降順で処理するよう指示します。**limit 10** オプションは、**foreach** に最初の 10 の反復のみを処理するよう指示します (つまり、値の高い上位 10 位の反復のみを出力します)。

3.5.5. アレイおよびアレイ要素の消去/削除

場合によっては、アレイ要素内の関連する値をクリアしたり、別のプローブで再利用するためにアレイ全体をリセットしたりする必要がある場合があります。「[アレイ内での複数要素の処理](#)」の例 3.15 「[cumulative-vfsreads.stp](#)」では、プロセスごとの VFS 読み取り数が時間の経過とともにどのように増加するかを追跡できますが、各プロセスが 3 秒間に行う VFS 読み取り数は表示されません。

これを実行するには、アレイが累積した値を消去する必要があります。**delete** 演算子を使用してアレイ内の要素またはアレイ全体を削除すると、これが実行できます。以下の例を見てみましょう。

例3.16 [noncumulative-vfsreads.stp](#)

```
global reads
probe vfs.read
{
  reads[execname()] ++
}
probe timer.s(3)
{
  foreach (count in reads)
    printf("%s : %d \n", count, reads[count])
  delete reads
}
```

例3.16 「[noncumulative-vfsreads.stp](#)」では、2つ目のプローブが調査した3秒間のみで各プロセスが実行したVFS reads 数を出力します。**delete reads** ステートメントは、プローブ内の **reads** アレイを消去します。

注記

同一プローブ内には、複数のアレイ演算を設置することが可能です。「[アレイ内での複数要素の処理](#)」および「[アレイおよびアレイ要素の消去/削除](#)」の例を使用すると、3秒間でプロセスが実行したVFS reads の数を追跡し、かつそれらのプロセスでの累積VFS reads 数を集計することができます。以下の例を考慮してください。

```
global reads, totalreads
probe vfs.read
{
  reads[execname()] ++
  totalreads[execname()] ++
}
probe timer.s(3)
{
  printf("=====\n")
  foreach (count in reads-)
    printf("%s : %d \n", count, reads[count])
  delete reads
}
probe end
{
  printf("TOTALS\n")
  foreach (total in totalreads-)
    printf("%s : %d \n", total, totalreads[total])
}
```

この例では、**reads** と **totalreads** のアレイが同じ情報を追跡し、同様の方式で出力します。唯一の違いは、**reads** は3秒ごとに消去されるのに対して、**totalreads** は増え続けるという点です。

3.5.6. 条件付きステートメントにおけるアレイの使用

連想アレイは **if** ステートメントでも使用することができます。これは、アレイ内の値が特定の条件に一致した場合にサブルーチンを実行するという場合に便利です。以下の例を見てみましょう。

例3.17 `vfsreads-print-if-1kb.stp`

```
global reads
probe vfs.read
{
  reads[execname()] ++
}
probe timer.s(3)
{
  printf("=====\n")
  foreach (count in reads-)
    if (reads[count] >= 1024)
      printf("%s : %dkB \n", count, reads[count]/1024)
```

```

else
    printf("%s : %dB \n", count, reads[count])
}

```

例3.17 「`vfsreads-print-if-1kb.stp`」では、全プロセス一覧と各プロセスが VFS read を実行した回数が 3 秒ごとに出力されます。プロセス名の関連する値が 1024 以上の場合、スクリプト内の `if` ステートメントがこれを変換し、**kB** で出力します。

メンバーシップのテスト

特定の一意のキーがアレイのメンバーかどうかをテストすることもできます。以下の例のように、`if` ステートメント内でアレイ内のメンバーシップを使用することも可能です。

```
if([index_expression] in array_name) statement
```

以下の例を見てみましょう。

例3.18 `vfsreads-stop-on-stapio2.stp`

```

global reads
probe vfs.read
{
    reads[execname()] ++
}
probe timer.s(3)
{
    printf("=====\n")
    foreach (count in reads+)
        printf("%s : %d \n", count, reads[count])
    if(["stapio"] in reads) {
        printf("stapio read detected, exiting\n")
        exit()
    }
}

```

`if(["stapio"] in reads)` ステートメントは、一意のキー **stapio** がアレイ **reads** に追加されたら、**stapio read detected, exiting** を出力するようにスクリプトに指示します。

3.5.7. 統計集計の計算

統計集計は、新規データを素早くかつ大量に累積 (集計ストリーム統計情報のみを保存) することが重要な場合に、数値的値の統計情報を収集するために使用されます。統計集計はグローバル変数またはアレイ内の要素として使用できます。

統計集計に値を追加するには、演算子 `<<< value` を使用します。

例3.19 `stat-aggregates.stp`

```

global reads
probe vfs.read
{

```



```
reads[execname()] <<< count
}
```

例3.19 「stat-aggregates.stp」では、演算子 `<<< count` が、`reads` アレイ内の対応する `execname()` の関連する値に `count` が返した数字を **保存** します。これらの値は **保存される** のであって、各一意のキーの関連する値に追加されたり、現行の関連する値に置き換わるものではありません。各一意のキー (`execname()`) に複数の関連する値があり、ハンドラーが実行するプローブで累積していると考えればよいでしょう。



注記

例3.19 「stat-aggregates.stp」のコンテキストでは、`count` は、仮想ファイルシステムに返された `execname()` が書き込んだデータ量を返します。

統計集計によって収集されたデータを抽出するには、構文形式 `@extractor(variable/array index expression)` を使用します。`extractor` は次の整数抽出のいずれかです。

count

変数やアレイのインデックス式に保存されたすべての値の数を返します。例3.19 「stat-aggregates.stp」のサンプルプローブでは、`@count(writes[execname()])` の式は、アレイ `writes` の各一意のキーに **保存されている値の数** を返します。

sum

変数やアレイのインデックス式に保存されたすべての値の合計を返します。例3.19 「stat-aggregates.stp」のサンプルプローブでは、`@sum(writes[execname()])` の式は、アレイ `writes` の各一意のキーに **保存されている値すべての合計** を返します。

min

変数やアレイのインデックス式に保存されたすべての値で最も小さいものを返します。

max

変数やアレイのインデックス式に保存されたすべての値で最も大きいものを返します。

avg

変数やアレイのインデックス式に保存されたすべての値の平均を返します。

統計集計を使用するには、複数のインデックス式 (最大 5 つ) を使用するアレイコンストラクトを構築することもできます。これは、プローブ中に追加のコンテキスト情報を捕捉する際に便利です。以下に例を示します。

例3.20 複数のアレイインデックス

```
global reads
probe vfs.read
{
  reads[execname(),pid()] <<< 1
}
probe timer.s(3)
{
```

```

foreach([var1,var2] in reads)
  printf("%s (%d) : %d \n", var1, var2, @count(reads[var1,var2]))
}

```

例3.20「複数のアレイインデックス」では、最初のプローブで各プロセスが実行した VFS read の数を追跡します。この例が他と異なる点は、このアレイは実行された read をプロセス名とそれに対応するプロセス ID の両方に関連付けしている点です。

例3.20「複数のアレイインデックス」の2つ目のプローブは、アレイ **reads** が収集した情報を処理して出力する方法を示しています。**foreach** ステートメントは、最初のプローブのアレイ **reads** の最初のインスタンスに含まれる変数 (**var1** および **var2**) と同じ数を使用している点に注意してください。

3.6. TAPSETS

Tapsets は、SystemTap スクリプトで使用する事前作成されたプローブおよび関数のライブラリーを形成するスクリプトです。ユーザーが SystemTap スクリプトを実行すると、SystemTap はスクリプトのプローブイベントとハンドラーを tapset ライブラリーに対して確認します。そして SystemTap はスクリプトを C 言語に変換する前に対応するプローブと関数を読み込みます (SystemTap で発生するセッションの情報は、「アーキテクチャー」を参照してください)。

SystemTap スクリプトと同様に、tapsets はファイル名拡張子 **.stp** を使用します。タップセットの標準ライブラリーは、デフォルトで **/usr/share/systemtap/tapset/** ディレクトリーにあります。ただし SystemTap スクリプトとは異なり、tapsets は直接実行するものではなく、他のスクリプトがそこから定義をプルできるライブラリーを設定します。

tapset ライブラリーは抽象化レイヤーで、ユーザーによるイベントおよび関数の定義を容易にするように設計されています。tapset は、ユーザーがイベントとして特定を希望する可能性のある関数に便利なエイリアスを提供します。適切なエイリアスを知ることは、ほとんどの場合、カーネルのバージョンごとに異なる特定のカーネル関数を覚えるよりも容易です。

「イベント」および SystemTap 関数のいくつかのハンドラーと関数は、tapset で定義されています。たとえば、**thread_indent()** は **indent.stp** で定義されています。

第4章 便利な SYSTEMTAP スクリプト

本章では、各種のサブシステムの監視および調査に使用可能な SystemTap スクリプトをいくつか説明します。これらのスクリプトはすべて、systemtap-testsuite パッケージをインストールすると、`/usr/share/systemtap/testsuite/systemtap.examples/` ディレクトリーで使用できます。

4.1. ネットワーク

以下のセクションでは、ネットワーク関連の関数を追跡し、ネットワークアクティビティーのプロファイルを構築するスクリプトを説明します。

4.1.1. ネットワークのプロファイリング

このセクションでは、ネットワークアクティビティーのプロファイルを実行する方法を説明します。例 4.1 「nettop.stp」では、マシン上で各プロセスが生成しているネットワークトラフィックの量が確認できます。

例4.1 nettop.stp

```
#!/usr/bin/env stap

global ifxmit, ifrecv
global ifmerged

probe netdev.transmit
{
  ifxmit[pid(), dev_name, execname(), uid()] <<< length
}

probe netdev.receive
{
  ifrecv[pid(), dev_name, execname(), uid()] <<< length
}

function print_activity()
{
  printf("%5s %5s %-7s %7s %7s %7s %7s %-15s\n",
    "PID", "UID", "DEV", "XMIT_PK", "RECV_PK",
    "XMIT_KB", "RECV_KB", "COMMAND")

  foreach ([pid, dev, exec, uid] in ifrecv) {
    ifmerged[pid, dev, exec, uid] += @count(ifrecv[pid,dev,exec,uid]);
  }
  foreach ([pid, dev, exec, uid] in ifxmit) {
    ifmerged[pid, dev, exec, uid] += @count(ifxmit[pid,dev,exec,uid]);
  }
  foreach ([pid, dev, exec, uid] in ifmerged-) {
    n_xmit = @count(ifxmit[pid, dev, exec, uid])
    n_recv = @count(ifrecv[pid, dev, exec, uid])
    printf("%5d %5d %-7s %7d %7d %7d %7d %-15s\n",
      pid, uid, dev, n_xmit, n_recv,
      n_xmit ? @sum(ifxmit[pid, dev, exec, uid])/1024 : 0,
      n_recv ? @sum(ifrecv[pid, dev, exec, uid])/1024 : 0,
      exec)
  }
}
```

```

}

print("\n")

delete ifxmit
delete ifrecv
delete ifmerged
}

probe timer.ms(5000), end, error
{
  print_activity()
}

```

`print_activity()` 関数は、以下の式を使用する点に注意してください。

```

n_xmit ? @sum(ifxmit[pid, dev, exec, uid])/1024 : 0
n_recv ? @sum(ifrecv[pid, dev, exec, uid])/1024 : 0

```

これらの式は、**if** または **else** 条件分岐です。2 番目のステートメントは、次の疑似コードをより簡潔に記述したものです。

```

if n_recv != 0 then
  @sum(ifrecv[pid, dev, exec, uid])/1024
else
  0

```

例4.1 「`nettop.stp`」では、どのプロセスがシステム上でネットワークトラフィックを生成しているかを追跡し、各プロセスについて以下の情報を提供します。

- **PID** – 一覧表示されているプロセスの ID。
- **UID** – ユーザー ID。ユーザー ID が **0** の場合は、root ユーザーを指します。
- **DEV** – データの送受信に使用されるイーサネットデバイス (eth0、eth1 など)。
- **XMIT_PK** – プロセスが送信したパケット数。
- **RECV_PK** – プロセスが受信したパケット数。
- **XMIT_KB** – プロセスが送信したデータ量 (キロバイト単位)。
- **RECV_KB** – サービスが受信したデータ量 (キロバイト単位)。

例4.1 「`nettop.stp`」では、5 秒ごとにネットワークプロファイルのサンプルが提供されます。この設定は、`probe timer.ms(5000)` を適切に編集すると、変更できます。例4.2 「例4.1 「`nettop.stp`」 出力サンプル」には、例4.1 「`nettop.stp`」からの 20 秒間の出力の抜粋が含まれています。

例4.2 例4.1 「`nettop.stp`」 出力サンプル

```

[...]
PID UID DEV XMIT_PK RECV_PK XMIT_KB RECV_KB COMMAND
  0  0 eth0    0    5    0    0 swapper
11178  0 eth0    2    0    0    0 synergyc

```

```

PID UID DEV XMIT_PK RECV_PK XMIT_KB RECV_KB COMMAND
2886 4 eth0 79 0 5 0 cups-polld
11362 0 eth0 0 61 0 5 firefox
0 0 eth0 3 32 0 3 swapper
2886 4 lo 4 4 0 0 cups-polld
11178 0 eth0 3 0 0 0 synergyc
PID UID DEV XMIT_PK RECV_PK XMIT_KB RECV_KB COMMAND
0 0 eth0 0 6 0 0 swapper
2886 4 lo 2 2 0 0 cups-polld
11178 0 eth0 3 0 0 0 synergyc
3611 0 eth0 0 1 0 0 Xorg
PID UID DEV XMIT_PK RECV_PK XMIT_KB RECV_KB COMMAND
0 0 eth0 3 42 0 2 swapper
11178 0 eth0 43 1 3 0 synergyc
11362 0 eth0 0 7 0 0 firefox
3897 0 eth0 0 1 0 0 multiload-apple
[...]
```

4.1.2. ネットワークソケットコードで呼び出された関数の追跡

このセクションでは、カーネルの `net/socket.c` ファイルから呼び出された関数を追跡する方法について説明します。このタスクでは、各プロセスがカーネルレベルでネットワークと対話する様子が詳細に分かります。

例4.3 socket-trace.stp

```

#!/usr/bin/stap

probe kernel.function("*@net/socket.c").call {
    printf ("%s -> %s\n", thread_indent(1), probefunc())
}
probe kernel.function("*@net/socket.c").return {
    printf ("%s <- %s\n", thread_indent(-1), probefunc())
}
```

例4.3 「socket-trace.stp」は、例3.6 「thread_indent.stp」と同じものです。これは以前、`thread_indent()` が機能する様子を説明するために `SystemTap` 関数 で使用されました。

例4.4 例4.3 「socket-trace.stp」出力サンプル

```

[...]
```

```

0 Xorg(3611): -> sock_poll
3 Xorg(3611): <- sock_poll
0 Xorg(3611): -> sock_poll
3 Xorg(3611): <- sock_poll
0 gnome-terminal(11106): -> sock_poll
5 gnome-terminal(11106): <- sock_poll
0 scim-bridge(3883): -> sock_poll
3 scim-bridge(3883): <- sock_poll
0 scim-bridge(3883): -> sys_socketcall
4 scim-bridge(3883): -> sys_recv
8 scim-bridge(3883): -> sys_recvfrom
```

```

12 scim-bridge(3883):-> sock_from_file
16 scim-bridge(3883):<- sock_from_file
20 scim-bridge(3883):-> sock_recvmsg
24 scim-bridge(3883):<- sock_recvmsg
28 scim-bridge(3883): <- sys_recvfrom
31 scim-bridge(3883): <- sys_recv
35 scim-bridge(3883): <- sys_socketcall
[...]
```

例4.4「例4.3「socket-trace.stp」出力サンプル」には、3秒間の例4.3「socket-trace.stp」の出力を引用してあります。**thread_indent()**が提供するこのスクリプトの出力に関する詳細情報は、[SystemTap 関数 例3.6「thread_indent.stp」](#)を参照してください。

4.1.3. 着信 TCP 接続の監視

このセクションでは、着信 TCP 接続の監視方法を説明します。このタスクは、承認されていない、疑わしい、さもなくば望ましくないネットワークアクセス要求をリアルタイムで特定する場合に便利です。

例4.5 tcp_connections.stp

```

#!/usr/bin/env stap

probe begin {
  printf("%6s %16s %6s %6s %16s\n",
         "UID", "CMD", "PID", "PORT", "IP_SOURCE")
}

probe kernel.function("tcp_accept").return?,
kernel.function("inet_csk_accept").return? {
  sock = $return
  if (sock != 0)
    printf("%6d %16s %6d %6d %16s\n", uid(), execname(), pid(),
         inet_get_local_port(sock), inet_get_ip_source(sock))
}
```

例4.5「tcp_connections.stp」の実行中は、システムが受け付けた着信 TCP 接続の以下の情報がリアルタイムで出力されます。

- 現在の **UID**
- **CMD** - 接続を受け付けるコマンド
- そのコマンドの **PID**
- 接続が使用するポート
- TCP 接続の発信元となる IP アドレス

例4.6 例4.5「tcp_connections.stp」出力サンプル

```

UID      CMD  PID  PORT  IP_SOURCE
0        sshd 3165  22   10.64.0.227
0        sshd 3165  22   10.64.0.227

```

4.1.4. カーネルでのネットワークパケットドロップの監視

Linux のネットワークスタックは、様々な理由でパケットを破棄する場合があります。一部の Linux カーネルには、パケットが破棄された場所を簡単に追跡するトレースポイント

`kernel.trace("kfree_skb")` が含まれています。例4.7「[dropwatch.stp](#)」は、`kernel.trace("kfree_skb")` を使用して、パケットの破棄を追跡します。スクリプトは、5 秒間隔ごとにパケットを破棄する場所を要約します。

例4.7 dropwatch.stp

```

#!/usr/bin/stap

#####
# Dropwatch.stp
# Author: Neil Horman <nhorman@redhat.com>
# An example script to mimic the behavior of the dropwatch utility
# http://fedorahosted.org/dropwatch
#####

# Array to hold the list of drop points we find
global locations

# Note when we turn the monitor on and off
probe begin { printf("Monitoring for dropped packets\n") }
probe end { printf("Stopping dropped packet monitor\n") }

# increment a drop counter for every location we drop at
probe kernel.trace("kfree_skb") { locations[$location] <<< 1 }

# Every 5 seconds report our drop locations
probe timer.sec(5)
{
  printf("\n")
  foreach (l in locations-) {
    printf("%d packets dropped at location %p\n",
      @count(locations[l]), l)
  }
  delete locations
}

```

`kernel.trace("kfree_skb")` は、カーネル内でネットワークパケットがドロップした場所を追跡します。`kernel.trace("kfree_skb")` には、解放されているバッファへのポインター (`$skb`) と、解放されているバッファのカーネルコード内での場所 (`$location`) という 2 つの引数があります。

`dropwatch.stp` スクリプトを 15 秒間実行すると、例4.8「[例4.7「dropwatch.stp」出力サンプル](#)」のような結果が出力されます。ここでは、トレースポイントアドレスと実際のアドレスのミス数が記載されています。

例4.8 例4.7 「dropwatch.stp」 出力サンプル

```
Monitoring for dropped packets
51 packets dropped at location 0xffffffff8024cd0f
2 packets dropped at location 0xffffffff8044b472
51 packets dropped at location 0xffffffff8024cd0f
1 packets dropped at location 0xffffffff8044b472
97 packets dropped at location 0xffffffff8024cd0f
1 packets dropped at location 0xffffffff8044b472
Stopping dropped packet monitor
```

パケットドロップの場所をより意味のあるものにするには、`/boot/System.map-$(uname -r)` ファイルを参照してください。このファイルには、各関数の開始アドレスが記載されており、[例4.8 「例4.7 「dropwatch.stp」 出力サンプル」](#) の出力のアドレスを特定の関数名にマップできます。次の `/boot/System.map-$(uname -r)` ファイルのスニペットを考えると、アドレス `0xffffffff8024cd0f` が関数 `unix_stream_rcvmsg` にマップされます。また、アドレス `0xffffffff8044b472` は関数 `arp_rcv` にマップされます。

```
[...]
ffffff8024c5cd T unlock_new_inode
ffffff8024c5da t unix_stream_sendmsg
ffffff8024c920 t unix_stream_rcvmsg
ffffff8024cea1 t udp_v4_lookup_longway
[...]
ffffff8044addc t arp_process
ffffff8044b360 t arp_rcv
ffffff8044b487 t parp_redo
ffffff8044b48c t arp_solicit
[...]
```

4.2. ディスク

以下のセクションでは、ディスクおよびI/O アクティビティを監視するスクリプトを説明します。

4.2.1. ディスク読み取り/書き込みトラフィックの要約

このセクションでは、どのプロセスが最も重いディスクの読み取り/書き込みをシステムに実行しているかを特定する方法を説明します。

例4.9 disktop.stp

```
#!/usr/bin/stap
#
# Copyright (C) 2007 Oracle Corp.
#
# Get the status of reading/writing disk every 5 seconds,
# output top ten entries
#
# This is free software,GNU General Public License (GPL);
# either version 2, or (at your option) any later version.
#
# Usage:
```



```

# ./disktop.stp
#

global io_stat,device
global read_bytes,write_bytes

probe vfs.read.return {
  if ($return>0) {
    if (devname!="N/A") {/*skip read from cache*/
      io_stat[pid(),execname(),uid(),ppid(),"R"] += $return
      device[pid(),execname(),uid(),ppid(),"R"] = devname
      read_bytes += $return
    }
  }
}

probe vfs.write.return {
  if ($return>0) {
    if (devname!="N/A") {/*skip update cache*/
      io_stat[pid(),execname(),uid(),ppid(),"W"] += $return
      device[pid(),execname(),uid(),ppid(),"W"] = devname
      write_bytes += $return
    }
  }
}

probe timer.ms(5000) {
  /* skip non-read/write disk */
  if (read_bytes+write_bytes) {

    printf("\n%-25s, %-8s%4dKb/sec, %-7s%6dKb, %-7s%6dKb\n\n",
      ctime(gettimeofday_s()),
      "Average:", ((read_bytes+write_bytes)/1024)/5,
      "Read:",read_bytes/1024,
      "Write:",write_bytes/1024)

    /* print header */
    printf("%8s %8s %8s %25s %8s %4s %12s\n",
      "UID","PID","PPID","CMD","DEVICE","T","BYTES")
  }
  /* print top ten I/O */
  foreach ([process,cmd,userid,parent,action] in io_stat- limit 10)
    printf("%8d %8d %8d %25s %8s %4s %12d\n",
      userid,process,parent,cmd,
      device[process,cmd,userid,parent,action],
      action,io_stat[process,cmd,userid,parent,action])

  /* clear data */
  delete io_stat
  delete device
  read_bytes = 0
  write_bytes = 0
}

probe end{
  delete io_stat

```

```

delete device
delete read_bytes
delete write_bytes
}

```

例4.9 「disktop.stp」 はディスクへの読み取りまたは書き込みが最も多い上位 10 のプロセスを出力します。例4.10 「例4.9 「disktop.stp」 出力サンプル」 は、このスクリプトの出力サンプルを示しており、リストされたプロセスごとに次のデータが含まれています。

- **UID** – ユーザー ID。ユーザー ID が **0** の場合は、root ユーザーを参照します。
- **PID** – 一覧表示されているプロセスの ID。
- **PPID** – 一覧表示されているプロセスの **親プロセス** のプロセス ID。
- **CMD** – 一覧表示されているプロセスの名前。
- **DEVICE** – 一覧表示されているプロセスが、読み取りまたは書き込みを行っているストレージデバイス。
- **T** – プロセスが実行したアクションのタイプ。**W** は書き込みを、**R** は読み取りを指します。
- **BYTES** – ディスクに対して読み書きされるデータの量。

例4.9 「disktop.stp」 の出力の時刻と日付は関数 `ctime()` と `gettimeofday_s()` によって戻されます。`ctime()` は Unix 時間の開始 (1970 年 1 月 1 日) からの経過秒数でカレンダー時間を導出します。`gettimeofday_s()` は Unix 時間の開始からの **実際** の秒数をカウントします。これにより、人間が判読できるかなり正確な出力のタイムスタンプが得られます。

このスクリプトでは、`$return` は各プロセスが仮想ファイルシステムから読み書きする実際のバイト数を保存するローカル変数です。`$return` は、return プローブ (`vfs.read.return`、`vfs.read.return` など) のみ使用できます。

例4.10 例4.9 「disktop.stp」 出力サンプル

```

[...]
Mon Sep 29 03:38:28 2008 , Average: 19Kb/sec, Read: 7Kb, Write: 89Kb
UID  PID  PPID          CMD DEVICE T BYTES
0  26319 26294        firefox sda5  W   90229
0  2758  2757    pam_timestamp_c sda5  R    8064
0  2885   1         cupsd sda5  W    1678
Mon Sep 29 03:38:38 2008 , Average: 1Kb/sec, Read: 7Kb, Write: 1Kb
UID  PID  PPID          CMD DEVICE T BYTES
0  2758  2757    pam_timestamp_c sda5  R    8064
0  2885   1         cupsd sda5  W    1678

```

4.2.2. ファイル読み取り/書き込みの I/O 時間の追跡

このセクションでは、各プロセスのファイルの読み取りおよび書き込み時間を監視する方法を説明します。これは、あるシステム上でどのファイルの読み込みが遅いかということを判断する際に便利です。

例4.11 iotime.stp

```
global start
global entry_io
global fd_io
global time_io

function timestamp:long() {
    return gettimeofday_us() - start
}

function proc:string() {
    return sprintf("%d (%s)", pid(), execname())
}

probe begin {
    start = gettimeofday_us()
}

global filenames
global filehandles
global fileread
global filewrite

probe syscall.open {
    filenames[pid()] = user_string($filename)
}

probe syscall.open.return {
    if ($return != -1) {
        filehandles[pid(), $return] = filenames[pid()]
        fileread[pid(), $return] = 0
        filewrite[pid(), $return] = 0
    } else {
        printf("%d %s access %s fail\n", timestamp(), proc(), filenames[pid()])
    }
    delete filenames[pid()]
}

probe syscall.read {
    if ($count > 0) {
        fileread[pid(), $fd] += $count
    }
    t = gettimeofday_us(); p = pid()
    entry_io[p] = t
    fd_io[p] = $fd
}

probe syscall.read.return {
    t = gettimeofday_us(); p = pid()
    fd = fd_io[p]
    time_io[p,fd] <<< t - entry_io[p]
}

probe syscall.write {
    if ($count > 0) {
        filewrite[pid(), $fd] += $count
    }
}
```

```

t = gettimeofday_us(); p = pid()
entry_io[p] = t
fd_io[p] = $fd
}

probe syscall.write.return {
t = gettimeofday_us(); p = pid()
fd = fd_io[p]
time_io[p,fd] <<< t - entry_io[p]
}

probe syscall.close {
if (filehandles[pid(), $fd] != "") {
printf("%d %s access %s read: %d write: %d\n", timestamp(), proc(),
filehandles[pid(), $fd], fileread[pid(), $fd], filewrite[pid(), $fd])
if (@count(time_io[pid(), $fd]))
printf("%d %s iotime %s time: %d\n", timestamp(), proc(),
filehandles[pid(), $fd], @sum(time_io[pid(), $fd]))
}
delete fileread[pid(), $fd]
delete filewrite[pid(), $fd]
delete filehandles[pid(), $fd]
delete fd_io[pid()]
delete entry_io[pid()]
delete time_io[pid(),$fd]
}
}

```

例4.11 「[iotime.stp](#)」では、システムコールが開かれる、閉じる、ファイルから読み取る、およびファイルに書き込む際に、毎回これを追跡します。例4.11 「[iotime.stp](#)」はシステムコールがアクセスする各ファイルについて、読み取りもしくは書き込みが終了するまでの時間をマイクロ秒単位でカウントし、読み取りもしくは書き込みされたデータ量をバイト単位で追跡します。

例4.11 「[iotime.stp](#)」は、ローカル変数 **\$count** を使用してシステムコールが読み取りまたは書き込みを試みたデータ量 (バイト単位) も追跡します。**\$return** (「[ディスク読み取り/書き込みトラフィックの要約](#)」の例4.9 「[disktop.stp](#)」で使用) には、読み書きされた **実際** のデータ量が保存されます。**\$count** は、データの読み取りまたは書き込みを追跡するプローブ (**syscall.read**、**syscall.write** など) でのみ使用できます。

例4.12 例4.11 「[iotime.stp](#)」 出力サンプル

```

[...]
825946 3364 (NetworkManager) access /sys/class/net/eth0/carrier read: 8190 write: 0
825955 3364 (NetworkManager) iotime /sys/class/net/eth0/carrier time: 9
[...]
117061 2460 (pcscd) access /dev/bus/usb/003/001 read: 43 write: 0
117065 2460 (pcscd) iotime /dev/bus/usb/003/001 time: 7
[...]
3973737 2886 (sendmail) access /proc/loadavg read: 4096 write: 0
3973744 2886 (sendmail) iotime /proc/loadavg time: 11
[...]

```

例4.12 「例4.11 「[iotime.stp](#)」 出力サンプル」では、以下のデータが出力されます。

- タイムスタンプ (マイクロ秒単位)
- プロセス ID およびプロセス名
- **access** または **iotime** のフラグ
- アクセスされたファイル

プロセスがデータの読み取りまたは書き込みを実行すると、**access** と **iotime** の行のペアと一緒に表示されます。**access** 行のタイムスタンプは、そのプロセスがファイルにアクセスを開始した時間を指します。その行の最後には、読み取り/書き込みされたデータ量がバイト単位で表示されます。**iotime** の行では、読み取り/書き込みにプロセスが費やした時間がマイクロ秒単位で表示されます。

access 行の後に **iotime** 行が続いていない場合は、プロセスがデータの読み取りまたは書き込みを行わなかったことを意味します。

4.2.3. 累積 I/O の追跡

このセクションでは、システムへの累積 I/O の量を追跡する方法を説明します。

例4.13 traceio.stp

```
#!/usr/bin/env stp
# traceio.stp
# Copyright (C) 2007 Red Hat, Inc., Eugene Teo <eteo@redhat.com>
# Copyright (C) 2009 Kai Meyer <kai@unixlords.com>
# Fixed a bug that allows this to run longer
# And added the humanreadable function
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License version 2 as
# published by the Free Software Foundation.
#

global reads, writes, total_io

probe vfs.read.return {
  reads[pid(),execname()] += $return
  total_io[pid(),execname()] += $return
}

probe vfs.write.return {
  writes[pid(),execname()] += $return
  total_io[pid(),execname()] += $return
}

function humanreadable(bytes) {
  if (bytes > 1024*1024*1024) {
    return sprintf("%d GiB", bytes/1024/1024/1024)
  } else if (bytes > 1024*1024) {
    return sprintf("%d MiB", bytes/1024/1024)
  } else if (bytes > 1024) {
    return sprintf("%d KiB", bytes/1024)
  } else {
    return sprintf("%d B", bytes)
  }
}
```

```

}

probe timer.s(1) {
  foreach([p,e] in total_io- limit 10)
    printf("%8d %15s r: %12s w: %12s\n",
           p, e, humanreadable(reads[p,e]),
           humanreadable(writes[p,e]))
  printf("\n")
  # Note we don't zero out reads, writes and total_io,
  # so the values are cumulative since the script started.
}

```

例4.13 「[traceio.stp](#)」 は、I/O トラフィックに応じた上位 10 位の実行可能ファイルを出力します。さらに、これらの上位 10 位の実行可能ファイルによる I/O 読み取りおよび書き込みの累積分量も追跡します。この情報は追跡され、1 秒間隔で降順で出力されます。

例4.13 「[traceio.stp](#)」 では、「[ディスク読み取り/書き込みトラフィックの要約](#)」の例 4.9 「[disktop.stp](#)」でも使用されているローカル変数 **\$return** も使用されていることに注意してください。

例4.14 例4.13 「[traceio.stp](#)」 出力サンプル

```

[...]
  Xorg r: 583401 KiB w:    0 KiB
  floaters r:   96 KiB w:  7130 KiB
multiload-apple r:  538 KiB w:  537 KiB
  sshd r:   71 KiB w:   72 KiB
pam_timestamp_c r:  138 KiB w:    0 KiB
  staprun r:   51 KiB w:   51 KiB
  snmpd r:   46 KiB w:    0 KiB
  pcscd r:   28 KiB w:    0 KiB
irqbalance r:   27 KiB w:    4 KiB
  cupsd r:    4 KiB w:   18 KiB
  Xorg r: 588140 KiB w:    0 KiB
  floaters r:   97 KiB w:  7143 KiB
multiload-apple r:  543 KiB w:  542 KiB
  sshd r:   72 KiB w:   72 KiB
pam_timestamp_c r:  138 KiB w:    0 KiB
  staprun r:   51 KiB w:   51 KiB
  snmpd r:   46 KiB w:    0 KiB
  pcscd r:   28 KiB w:    0 KiB
irqbalance r:   27 KiB w:    4 KiB
  cupsd r:    4 KiB w:   18 KiB

```

4.2.4. I/O 監視 (デバイスごと)

このセクションでは、特定のデバイス上の I/O アクティビティを監視する方法を説明します。

例4.15 [traceio2.stp](#)

```

#!/usr/bin/env stap

```

```

global device_of_interest

probe begin {
    /* The following is not the most efficient way to do this.
       One could directly put the result of usrdev2kerndev()
       into device_of_interest. However, want to test out
       the other device functions */
    dev = usrdev2kerndev($1)
    device_of_interest = MKDEV(MAJOR(dev), MINOR(dev))
}

probe vfs.write, vfs.read
{
    if (dev == device_of_interest)
        printf ("%s(%d) %s 0x%x\n",
                execname(), pid(), probefunc(), dev)
}

```

例4.15 「traceio2.stp」は、全体デバイス番号という引数1つを取ります。この番号を取得するには、**stat -c "0x%D" directory** を使用します。この *directory* は、監視するデバイス上にあります。

usrdev2kerndev() 関数は、全体デバイス番号をカーネルが理解する書式に変換します。**usrdev2kerndev()** が生成した出力は、特定デバイスのメジャー番号とマイナー番号を決定するために **MKDEV()**、**MINOR()**、および **MAJOR()** 関数とともに使用されます。

例4.15 「traceio2.stp」の出力には、読み取り/書き込みを実行しているプロセスの名前と ID、実行している関数 (**vfs_read** または **vfs_write**)、およびカーネルデバイス番号が含まれます。

次の例は、**stap traceio2.stp 0x805** の完全な出力からの抜粋です。ここで、**0x805** は **/home** のデバイス番号全体です。**/home** は、監視したいデバイスである **/dev/sda5** にあります。

例4.16 例4.15 「traceio2.stp」出力サンプル

```

[...]
synergyc(3722) vfs_read 0x800005
synergyc(3722) vfs_read 0x800005
cupsd(2889) vfs_write 0x800005
cupsd(2889) vfs_write 0x800005
cupsd(2889) vfs_write 0x800005
[...]

```

4.2.5. ファイルの読み取りおよび書き込みの監視

このセクションでは、ファイルの読み取りおよび書き込みをリアルタイムで監視する方法を説明します。

例4.17 inodewatch.stp

```

#!/usr/bin/env stap

probe vfs.write, vfs.read
{

```

```
# dev and ino are defined by vfs.write and vfs.read
if (dev == MKDEV($1,$2) # major/minor device
    && ino == $3)
    printf ("%s(%d) %s 0x%x/%u\n",
        execname(), pid(), probefunc(), dev, ino)
}
```

例4.17 「`inodewatch.stp`」 は、ファイルの以下の情報をコマンドラインの引数として取ります。

- ファイルのメジャーデバイス番号
- ファイルのマイナーデバイス番号
- ファイルの **inode** 番号

この情報を取得するには、`stat -c '%D %i' filename` を使用します。 *filename* は絶対パスです。

たとえば、`/etc/crontab` ファイルを監視する場合は、最初に `stat -c '%D %i' /etc/crontab` を実行します。これにより、次の出力が出力されます。

```
805 1078319
```

805 は、ベース 16 (16 進数) のデバイス番号です。最後の 2 桁がマイナーデバイス番号で、その上の数字がメジャー番号です。**1078319** は **inode** 番号です。`/etc/crontab` の監視を開始するには、`stap inodewatch.stp 0x8 0x05 1078319` を実行します (**0x** 接頭辞は base-16 値を示します)。

このコマンドの出力には、読み取り/書き込みを実行しているプロセスの名前と ID、実行中の機能 (`vfs_read` または `vfs_write`)、デバイス番号 (16 進形式)、および **inode** 番号が含まれます。例4.18 「例4.17 「`inodewatch.stp`」 出力サンプル」には、`stap inodewatch.stp 0x8 0x05 1078319` の出力が含まれます (スクリプトの実行中に `cat /etc/crontab` が実行された場合)。

例4.18 例4.17 「`inodewatch.stp`」 出力サンプル

```
cat(16437) vfs_read 0x800005/1078319
cat(16437) vfs_read 0x800005/1078319
```

4.2.6. ファイル属性の変更の監視

このセクションでは、プロセスがターゲットファイルの属性を変更したかどうかをリアルタイムで監視する方法を説明します。

例4.19 `inodewatch2-simple.stp`

```
global ATTR_MODE = 1

probe kernel.function("inode_setattr") {
    dev_nr = $inode->i_sb->s_dev
    inode_nr = $inode->i_ino

    if (dev_nr == ($1 << 20 | $2) # major/minor device
        && inode_nr == $3
        && $attr->ia_valid & ATTR_MODE)
```



```

printf ("%s(%d) %s 0x%x/%u %o %d\n",
        execname(), pid(), probefunc(), dev_nr, inode_nr, $attr->ia_mode, uid())
}

```

「ファイルの読み取りおよび書き込みの監視」の例4.17「inodewatch.stp」のように、例4.19「inodewatch2-simple.stp」はターゲットファイルのデバイス番号(整数形式)と **inode** 番号を引数として取ります。この情報の取得方法は、「ファイルの読み取りおよび書き込みの監視」を参照してください。

例4.19「inodewatch2-simple.stp」の出力は例4.17「inodewatch.stp」の出力に似ていますが、例4.19「inodewatch2-simple.stp」には監視対象ファイルの属性変更と、変更を行ったユーザー ID (**uid()**) が含まれている点で異なります。例4.20「例4.19「inodewatch2-simple.stp」出力サンプル」は例4.19「inodewatch2-simple.stp」の出力になります。**/home/joe/bigfile** の監視中に、ユーザー **joe** が **chmod 777 /home/joe/bigfile** と **chmod 666 /home/joe/bigfile** を実行しています。

例4.20 例4.19「inodewatch2-simple.stp」出力サンプル

```

chmod(17448) inode_setattr 0x800005/6011835 100777 500
chmod(17449) inode_setattr 0x800005/6011835 100666 500

```

4.3. プロファイリング

以下のセクションでは、関数呼び出しを監視することでカーネルアクティビティのプロファイルを実行するスクリプトを説明します。

4.3.1. 関数呼び出しのカウンタ

このセクションでは、30秒間のサンプルで特定のカーネル関数をシステムが呼び出した回数を特定する方法を説明します。ワイルドカードの使用によっては、このスクリプトを使用して複数のカーネル関数を対象とすることもできます。

例4.21 functioncallcount.stp

```

#!/usr/bin/env stap
# The following line command will probe all the functions
# in kernel's memory management code:
#
# stap functioncallcount.stp "@mm/*.c"

probe kernel.function(@1).call { # probe functions listed on commandline
    called[probefunc()] <<< 1 # add a count efficiently
}

global called

probe end {
    foreach (fn in called-) # Sort by call count (in decreasing order)
    # (fn+ in called) # Sort by function name
        printf("%s %d\n", fn, @count(called[fn]))
    exit()
}

```

例4.21 「`functioncallcount.stp`」は、ターゲットカーネルの関数を引数として取ります。この引数はワイルドカードに対応しているので、ある程度までの複数のカーネル関数をターゲットにできます。

例4.21 「`functioncallcount.stp`」の出力には、サンプル時間中に呼び出された関数の名前とその回数がアルファベット順に表示されます。例4.22 「例4.21 「`functioncallcount.stp`」出力サンプル」は、`stap functioncallcount.stp "*"@mm/*.c`の出力抜粋になります。

例4.22 例4.21 「`functioncallcount.stp`」出力サンプル

```
[...]
__vma_link 97
__vma_link_file 66
__vma_link_list 97
__vma_link_rb 97
__xchg 103
add_page_to_active_list 102
add_page_to_inactive_list 19
add_to_page_cache 19
add_to_page_cache_lru 7
all_vm_events 6
alloc_pages_node 4630
alloc_slabmgmt 67
anon_vma_alloc 62
anon_vma_free 62
anon_vma_lock 66
anon_vma_prepare 98
anon_vma_unlink 97
anon_vma_unlock 66
arch_get_unmapped_area_topdown 94
arch_get_unmapped_exec_area 3
arch_unmap_area_topdown 97
atomic_add 2
atomic_add_negative 97
atomic_dec_and_test 5153
atomic_inc 470
atomic_inc_and_test 1
[...]
```

4.3.2. 呼び出し先の追跡

このセクションでは、着信および発信関数呼び出しを追跡する方法を説明します。

例4.23 `para-callgraph.stp`

```
#!/usr/bin/env stap

function trace(entry_p, extra) {
  %( $# > 1 %? if (tid() in trace) %)
  printf("%s%s%s %s\n",
    thread_indent (entry_p),
    (entry_p>0?"->":"<-"),
    probefunc ()),
```

```

        extra)
    }

    %( $# > 1 %?
    global trace
    probe $2.call {
        trace[tid()] = 1
    }
    probe $2.return {
        delete trace[tid()]
    }
    %)

    probe $1.call { trace(1, $$parms) }
    probe $1.return { trace(-1, $$return) }

```

例4.23 「`para-callgraph.stp`」 は、以下の2つのコマンドライン引数を取っています。

- その開始または終了が追跡対象となっている関数 (**\$1**)。
- 2つ目のオプションとなる *trigger function* (**\$2**)。これは、スレッドごとの追跡を有効、無効にします。trigger function が終了していなければ、各スレッドにおける追跡は継続されます。

例4.23 「`para-callgraph.stp`」 では `thread_indent()` を使用しているので、その出力には、**\$1** (追跡しているプローブ関数) のタイムスタンプ、プロセス名、およびスレッド ID が含まれます。 `thread_indent()` の詳細は、 [SystemTap 関数](#) のエントリを参照してください。

次の例には、 `stap para-callgraph.stp 'kernel.function("*@fs/*.c") 'kernel.function("sys_read")'` の出力からの抜粋が含まれています。

例4.24 例4.23 「`para-callgraph.stp`」 出力サンプル

```

[...]
267 gnome-terminal(2921): <-do_sync_read return=0xffffffffffff5
269 gnome-terminal(2921):<-vfs_read return=0xffffffffffff5
0 gnome-terminal(2921):->fput file=0xffff880111ebbc0
2 gnome-terminal(2921):<-fput
0 gnome-terminal(2921):->fget_light fd=0x3 fput_needed=0xffff88010544df54
3 gnome-terminal(2921):<-fget_light return=0xffff8801116ce980
0 gnome-terminal(2921):->vfs_read file=0xffff8801116ce980 buf=0xc86504 count=0x1000
pos=0xffff88010544df48
4 gnome-terminal(2921): ->rw_verify_area read_write=0x0 file=0xffff8801116ce980
ppos=0xffff88010544df48 count=0x1000
7 gnome-terminal(2921): <-rw_verify_area return=0x1000
12 gnome-terminal(2921): ->do_sync_read filp=0xffff8801116ce980 buf=0xc86504 len=0x1000
ppos=0xffff88010544df48
15 gnome-terminal(2921): <-do_sync_read return=0xffffffffffff5
18 gnome-terminal(2921):<-vfs_read return=0xffffffffffff5
0 gnome-terminal(2921):->fput file=0xffff8801116ce980

```

4.3.3. カーネルおよびユーザースペースで費やした時間の判定

このセクションでは、スレッドがカーネルまたはユーザースペースで費やした時間を判定する方法を説明します。

例4.25 thread-times.stp

```
#!/usr/bin/env stap

probe perf.sw.cpu_clock!, timer.profile {
  // NB: To avoid contention on SMP machines, no global scalars/arrays used,
  // only contention-free statistics aggregates.
  tid=tid(); e=execname()
  if (!user_mode())
    kticks[e,tid] <<< 1
  else
    uticks[e,tid] <<< 1
  ticks <<< 1
  tids[e,tid] <<< 1
}

global uticks, kticks, ticks

global tids

probe timer.s(5), end {
  allticks = @count(ticks)
  printf ("%16s %5s %7s %7s (of %d ticks)\n",
    "comm", "tid", "%user", "%kernel", allticks)
  foreach ([e,tid] in tids- limit 20) {
    uscaled = @count(uticks[e,tid])*10000/allticks
    // SystemTap only performs integer arithmetic.
    // To avoid losing precision the decimal point is shifted
    // to the right four places (*100000). Think of this as
    // the original result value x.xxyy becoming xxxyy.0.
    // The integer percentage xxx is obtained
    // by dividing by 100 and the fractional percentage
    // is obtained with a modulo 100 operation.
    kscaled = @count(kticks[e,tid])*10000/allticks
    printf ("%16s %5d %3d.%02d%% %3d.%02d%%\n",
      e, tid, uscaled/100, uscaled%100, kscaled/100, kscaled%100)
  }
  printf("\n")

  delete uticks
  delete kticks
  delete ticks
  delete tids
}
```

例4.25 「thread-times.stp」 は、5 秒間のサンプル内で CPU 時間を占めている上位 20 位のプロセスと、CPU ティックの総数を一覧表示します。このスクリプトの出力は、各プロセスが使用した CPU 時間のパーセント表示と、その時間がカーネルスペースかユーザースペースで費やされたかも示します。

例4.26 「例4.25 「thread-times.stp」 出力サンプル」 は、例4.25 「thread-times.stp」 の 5 秒間サンプルの出力です。

例4.26 例4.25 「thread-times.stp」 出力サンプル

```

tid %user %kernel (of 20002 ticks)
 0 0.00% 87.88%
32169 5.24% 0.03%
9815 3.33% 0.36%
9859 0.95% 0.00%
3611 0.56% 0.12%
9861 0.62% 0.01%
11106 0.37% 0.02%
32167 0.08% 0.08%
3897 0.01% 0.08%
3800 0.03% 0.00%
2886 0.02% 0.00%
3243 0.00% 0.01%
3862 0.01% 0.00%
3782 0.00% 0.00%
21767 0.00% 0.00%
2522 0.00% 0.00%
3883 0.00% 0.00%
3775 0.00% 0.00%
3943 0.00% 0.00%
3873 0.00% 0.00%

```

4.3.4. ポーリングアプリケーションの監視

このセクションでは、どのアプリケーションがポーリングを行なっているかを特定、監視する方法を説明します。これを行うことで、不必要または過剰なポーリングが追跡でき、CPUの使用量を改善し、省電力効果を向上させるエリアを特定できるようになります。

例4.27 timeout.stp

```

#!/usr/bin/env stap
# Copyright (C) 2009 Red Hat, Inc.
# Written by Ulrich Drepper <drepper@redhat.com>
# Modified by William Cohen <wcohen@redhat.com>

global process, timeout_count, to
global poll_timeout, epoll_timeout, select_timeout, itimer_timeout
global nanosleep_timeout, futex_timeout, signal_timeout

probe syscall.poll, syscall.epoll_wait {
    if (timeout) to[pid()]=timeout
}

probe syscall.poll.return {
    p = pid()
    if ($return == 0 && to[p] > 0) {
        poll_timeout[p]++
        timeout_count[p]++
        process[p] = execname()
        delete to[p]
    }
}

```

```
}

probe syscall.epoll_wait.return {
  p = pid()
  if ($return == 0 && to[p] > 0 ) {
    epoll_timeout[p]++
    timeout_count[p]++
    process[p] = execname()
    delete to[p]
  }
}

probe syscall.select.return {
  if ($return == 0) {
    p = pid()
    select_timeout[p]++
    timeout_count[p]++
    process[p] = execname()
  }
}

probe syscall.futex.return {
  if (errno_str($return) == "ETIMEDOUT") {
    p = pid()
    futex_timeout[p]++
    timeout_count[p]++
    process[p] = execname()
  }
}

probe syscall.nanosleep.return {
  if ($return == 0) {
    p = pid()
    nanosleep_timeout[p]++
    timeout_count[p]++
    process[p] = execname()
  }
}

probe kernel.function("it_real_fn") {
  p = pid()
  itimer_timeout[p]++
  timeout_count[p]++
  process[p] = execname()
}

probe syscall.rt_sigtimedwait.return {
  if (errno_str($return) == "EAGAIN") {
    p = pid()
    signal_timeout[p]++
    timeout_count[p]++
    process[p] = execname()
  }
}

probe syscall.exit {
```

```

p = pid()
if (p in process) {
  delete process[p]
  delete timeout_count[p]
  delete poll_timeout[p]
  delete epoll_timeout[p]
  delete select_timeout[p]
  delete itimer_timeout[p]
  delete futex_timeout[p]
  delete nanosleep_timeout[p]
  delete signal_timeout[p]
}
}

probe timer.s(1) {
  ansi_clear_screen()
  printf (" pid | poll select  epoll itimer  futex nanosle  signal| process\n")
  foreach (p in timeout_count- limit 20) {
    printf ("%5d |%7d %7d %7d %7d %7d %7d %7d | %-.38s\n", p,
      poll_timeout[p], select_timeout[p],
      epoll_timeout[p], itimer_timeout[p],
      futex_timeout[p], nanosleep_timeout[p],
      signal_timeout[p], process[p])
  }
}

```

例4.27 「timeout.stp」 は、各アプリケーションが以下のシステムコールを使用した回数を追跡します。

- poll
- select
- epoll
- itimer
- futex
- nanosleep
- signal

いくつかのアプリケーションでは、これらのシステムコールは過剰に使用されています。このため、これらは通常、ポーリングアプリケーションの有力な容疑者として特定されます。ただし、アプリケーションが異なるシステムコールを使用して過剰にポーリングしている場合もあります。場合によっては、システムが使用している上位のシステムコールを見つけることが便利なこともあります(手順は「最もよく使われるシステムコールの追跡」を参照してください)。これを行うことで新たな問題を特定でき、例4.27 「timeout.stp」 に追加して追跡することができます。

例4.28 例4.27 「timeout.stp」 出力サンプル

```

uid | poll select  epoll itimer  futex nanosle  signal| process
28937 | 148793  0  0  4727  37288  0  0| firefox
22945 |  0 56949  0  1  0  0  0| scim-bridge

```

```

0 | 0 0 0 36414 0 0 0 | swapper
4275 | 23140 0 0 1 0 0 0 | mixer_applet2
4191 | 0 14405 0 0 0 0 0 | scim-launcher
22941 | 7908 1 0 62 0 0 0 | gnome-terminal
4261 | 0 0 0 2 0 7622 0 | escd
3695 | 0 0 0 0 0 7622 0 | gdm-binary
3483 | 0 7206 0 0 0 0 0 | dhcdbd
4189 | 6916 0 0 2 0 0 0 | scim-panel-gtk
1863 | 5767 0 0 0 0 0 0 | iscsid
2562 | 0 2881 0 1 0 1438 0 | pcscd
4257 | 4255 0 0 1 0 0 0 | gnome-power-man
4278 | 3876 0 0 60 0 0 0 | multiloader-apple
4083 | 0 1331 0 1728 0 0 0 | Xorg
3921 | 1603 0 0 0 0 0 0 | gam_server
4248 | 1591 0 0 0 0 0 0 | nm-applet
3165 | 0 1441 0 0 0 0 0 | xterm
29548 | 0 1440 0 0 0 0 0 | httpd
1862 | 0 0 0 0 0 1438 0 | iscsid

```

2つ目のプローブのタイマー (`timer.s()`) を編集すると、サンプル時間を長くすることができます。例 4.21 「[functioncallcount.stp](#)」 の出力には、上位 20 位のポーリングアプリケーションの名前と UID、各アプリケーションがシステムコールのポーリングを行った回数が含まれています。例 4.28 「例 4.27 「[timeout.stp](#)」 出力サンプル」 はスクリプトの抜粋になります。

4.3.5. 最もよく使われるシステムコールの追跡

「[ポーリングアプリケーションの監視](#)」 の例 4.27 「[timeout.stp](#)」 では、以下のシステムコールを最もよく使用したものを挙げることで、どのアプリケーションがポーリングを行なっているかを特定します。

- `poll`
- `select`
- `epoll`
- `itimer`
- `futex`
- `nanosleep`
- `signal`

ただしシステムによっては、別のシステムコールが過剰なポーリングを行なっている可能性もあります。ポーリングしているアプリケーションがポーリングに異なるシステムコールを使用していることが疑われる場合は、まずシステムが使用している上位のシステムコールを特定する必要があります。これには、例 4.29 「[topsys.stp](#)」 を使用します。

例 4.29 topsys.stp

```

#!/usr/bin/env stap
#
# This script continuously lists the top 20 systemcalls in the interval

```



```

# 5 seconds
#

global syscalls_count

probe syscall.* {
  syscalls_count[name]++
}

function print_systop () {
  printf ("%25s %10s\n", "SYSCALL", "COUNT")
  foreach (syscall in syscalls_count- limit 20) {
    printf ("%25s %10d\n", syscall, syscalls_count[syscall])
  }
  delete syscalls_count
}

probe timer.s(5) {
  print_systop ()
  printf("-----\n")
}

```

例4.29 「topsys.stp」は、5秒ごとにシステムが使用している上位20位のシステムコールを一覧表示します。また、同期間に各システムコールが使用された回数も表示されます。例4.30「例4.29「topsys.stp」出力サンプル」が出力サンプルになります。

例4.30 例4.29「topsys.stp」出力サンプル

```

-----
      SYSCALL  COUNT
gettimeofday  1857
      read    1821
      ioctl   1568
      poll    1033
      close   638
      open    503
      select  455
      write   391
      writev  335
      futex   303
      recvmsg 251
      socket  137
      clock_gettime 124
      rt_sigprocmask 121
      sendto  120
      setitimer 106
      stat    90
      time    81
      sigreturn 72
      fstat   66
-----

```

4.3.6. プロセスごとのシステムコールボリュームの追跡

このセクションでは、最もボリュームの大きいシステムコールを実行しているプロセスを判定する方法を説明します。ここまでのセクションでは、システムが使用している上位のシステムコールを監視する方法を説明しました(「[最もよく使われるシステムコールの追跡](#)」)。また、どのアプリケーションが最も多く特定のポーリング容疑者のシステムコールを使用しているかを特定する方法も説明しました(「[ポーリングアプリケーションの監視](#)」)。プロセスごとのシステムコールのボリュームを監視することで、ポーリングプロセスおよび他のリソースの大量消費についてシステムを調査する際により多くのデータが提供できます。

例4.31 syscalls_by_proc.stp

```
#!/usr/bin/env stap

# Copyright (C) 2006 IBM Corp.
#
# This file is part of systemtap, and is free software. You can
# redistribute it and/or modify it under the terms of the GNU General
# Public License (GPL); either version 2, or (at your option) any
# later version.

#
# Print the system call count by process name in descending order.
#

global syscalls

probe begin {
  print ("Collecting data... Type Ctrl-C to exit and display results\n")
}

probe syscall.* {
  syscalls[execname()]++
}

probe end {
  printf ("%10s %-s\n", "#SysCalls", "Process Name")
  foreach (proc in syscalls-)
    printf ("%10d %-s\n", syscalls[proc], proc)
}
```

例4.31 「[syscalls_by_proc.stp](#)」 は、システムコールを多く実行している上位 20 位のプロセスを一覧表示します。また、一定期間内に各プロセスが実行したシステムコールの数も表示されます。例4.32 「例4.29 「[topsys.stp](#)」 出力サンプル」 が出力サンプルになります。

例4.32 例4.29 「[topsys.stp](#)」 出力サンプル

```
Collecting data... Type Ctrl-C to exit and display results
#SysCalls Process Name
1577    multiload-apple
692     synergyc
408     pcscd
376     mixer_applet2
299     gnome-terminal
```

```

293   Xorg
206   scim-panel-gtk
95    gnome-power-man
90    artsd
85    dhcdbd
84    scim-bridge
78    gnome-screensav
66    scim-launcher
[...]

```

プロセス名の代わりにプロセス ID を表示するには、以下のスクリプトを使用します。

例4.33 syscalls_by_pid.stp

```

#!/usr/bin/env stap

# Copyright (C) 2006 IBM Corp.
#
# This file is part of systemtap, and is free software.  You can
# redistribute it and/or modify it under the terms of the GNU General
# Public License (GPL); either version 2, or (at your option) any
# later version.

#
# Print the system call count by process ID in descending order.
#

global syscalls

probe begin {
  print ("Collecting data... Type Ctrl-C to exit and display results\n")
}

probe syscall.* {
  syscalls[pid()]++
}

probe end {
  printf ("%10s %-s\n", "#SysCalls", "PID")
  foreach (pid in syscalls-)
    printf ("%10d %-d\n", syscalls[pid], pid)
}

```

出力で表示されているように、結果を表示するにはスクリプトを手動で終了する必要があります。どちらのスクリプトにも **timer.s()** プローブを追加して、一定時間後に終了させることができます。たとえば、5 秒後にスクリプトを終了させるには、以下のプローブをスクリプトに追加します。

```

probe timer.s(5)
{
  exit()
}

```

4.4. 競合ユーザースペースのロックの特定

このセクションでは、特定期間におけるシステムを通じた競合ユーザースペースのロックを特定する方法を説明します。競合ユーザースペースのロックの特定は、**futex** 競合が原因と疑われるハングの調査に役立ちます。

簡単に説明すると、複数のプロセスがメモリーの同じ領域にアクセスしようとする、**futex** 競合が発生します。場合によっては、これは競合しているプロセス間のデッドロックになり、アプリケーションがハングしているように見えます。

これを特定するために、[例4.34 「futexes.stp」](#) では **futex** システムコールをプローブします。

例4.34 futexes.stp

```
#!/usr/bin/env stap

# This script tries to identify contended user-space locks by hooking
# into the futex system call.

global thread_thislock # short
global thread_blocktime #
global FUTEX_WAIT = 0 /*, FUTEX_WAKE = 1 */

global lock_waits # long-lived stats on (tid,lock) blockage elapsed time
global process_names # long-lived pid-to-execname mapping

probe syscall.futex {
  if (op != FUTEX_WAIT) next # don't care about WAKE event originator
  t = tid ()
  process_names[pid()] = execname()
  thread_thislock[t] = $uaddr
  thread_blocktime[t] = gettimeofday_us()
}

probe syscall.futex.return {
  t = tid()
  ts = thread_blocktime[t]
  if (ts) {
    elapsed = gettimeofday_us() - ts
    lock_waits[pid(), thread_thislock[t]] <<< elapsed
    delete thread_blocktime[t]
    delete thread_thislock[t]
  }
}

probe end {
  foreach ([pid+, lock] in lock_waits)
    printf ("%s[%d] lock %p contended %d times, %d avg us\n",
            process_names[pid], pid, lock, @count(lock_waits[pid,lock]),
            @avg(lock_waits[pid,lock]))
}
```

[例4.34 「futexes.stp」](#) は手動で停止する必要があります。終了時には以下の情報が出力されます。

- 競合の原因となったプロセスの名前と ID
- 競合対象となったメモリー領域
- メモリー領域が競合された回数
- プローブ中の競合の平均時間

例4.35 「例4.34 「[futexes.stp](#)」 出力サンプル」 には、スクリプトの終了時 (約 20 秒後) に 例4.34 「[futexes.stp](#)」 の出力の抜粋が含まれます。

例4.35 例4.34 「[futexes.stp](#)」 出力サンプル

```
[...]
automount[2825] lock 0x00bc7784 contended 18 times, 999931 avg us
synergyc[3686] lock 0x0861e96c contended 192 times, 101991 avg us
synergyc[3758] lock 0x08d98744 contended 192 times, 101990 avg us
synergyc[3938] lock 0x0982a8b4 contended 192 times, 101997 avg us
[...]
```

第5章 SYSTEMTAP のエラーを理解する

この章では、SystemTap の使用中によく発生するエラーについて説明します。

多くの診断エラーには、次のような man ページを読むことを提案する文字列が含まれます。

```
[man warning::example1]
[man error::example2]
```

このような場合は、その診断に固有の説明情報を提供する **warning:: example1** および **error:: example2** のそれぞれの man ページを参照してください。

5.1. 解析エラーとセマンティックエラー

このタイプのエラーは、スクリプトがカーネルモジュールに変換される前の、SystemTap により構文解析が行われ C に変換される際に発生します。たとえば、これらのタイプのエラーは、無効な値を変数やアレイに割り当てる演算から発生します。

parse error: expected abc, saw xyz

スクリプトに文法的なエラーまたは誤字や脱字のエラーが含まれています。プローブのコンテキストから考えると、SystemTap は間違ったコンストラクトのタイプを検出しました。

以下の無効な SystemTap スクリプトにはプローブのハンドラーがありません。

```
probe vfs.read
probe vfs.write
```

その結果、パーサーは 2 行目のコラム 1 の **probe** キーワードには別の値を期待していたことを示す以下のエラーメッセージが表示されます。

```
parse error: expected one of '., (?!{ = +='
saw: keyword at perror.stp:2:1
1 parse error(s).
```

parse error: embedded code in unprivileged script

スクリプトには危険な埋め込み C コードが含まれています (%{ %} で囲まれたコードブロック)。SystemTap では、スクリプトに C コードを埋め込むことができるため、目的にあった tapsets がない場合には便利です。ただし、埋め込みの C コンストラクトは安全ではありません。そのため、スクリプトにこのような構造を検出すると SystemTap はこのエラーで警告を発します。

スクリプト内の似たようなコンストラクトがすべて確実に安全であることが分かっている、かつ **stapdev** グループのメンバーである (または root 特権がある) 場合には、オプション **-g** を使用して **guru** モードでスクリプトを実行します。

```
stap -g script
```

semantic error: type mismatch for identifier 'ident' ... string vs. long

スクリプト内の **ident** 関数は、間違ったタイプ (%s または %d) を使用していました。このエラーは [例 5.1 「error-variable.stp」](#) に表示されます。関数 **execname()** が文字列を返すため、書式指定子は %d ではなく %s にしてください。

例5.1 error-variable.stp

```
probe syscall.open
{
  printf ("%d(%d) open\n", execname(), pid())
}
```

semantic error: unresolved type for identifier 'ident'

識別子 (変数など) が使用されているがタイプ (整数または文字列) を特定できません。たとえば、**printf** ステートメントに変数を使用しているのに、スクリプトでは変数に値を割り当てていない場合などにこのエラーが発生します。

semantic error: Expecting symbol or array index expression

SystemTap ではアレイ内の場所や変数に値を割り当てることはできません。割り当ての目的地が有効な目的地になっていません。以下のようなサンプルコードを使用すると、このエラーが発生することになります。

```
probe begin { printf("x") = 1 }
```

semantic error: unresolved function call (変数 N 関数の検索中に発生するエラー)

スクリプト内の関数呼び出しまたはアレイインデックス式が無効な数の引数/パラメーターを使用しました。SystemTap では、*arity* (アリティ) はアレイのインデックス数または関数へのパラメーター数を参照することができます。

semantic error: array locals not supported, missing global declaration?

アレイをグローバル変数と宣言せずに、スクリプトがアレイ演算を使用しました (グローバル変数は、SystemTap スクリプトでの使用後に宣言されることができません)。アレイが一貫性のないアリティと使用されると、同様のメッセージが表示されます。

semantic error: variable 'vaar' modified during 'foreach' iteration

アレイ **var** がアクティブな **foreach** ループ内で修正 (割り当てまたは削除) されています。このエラーは、スクリプト内の演算が **foreach** ループ内で関数呼び出しを実行しても表示されます。

semantic error: probe point mismatch at position N, while resolving probe point pnt

SystemTap は、イベントまたは SystemTap 関数 **pnt** が参照しているものを理解できませんでした。これは通常、SystemTap が tapset ライブラリー内で **pnt** に一致するものを見つけられなかったことを意味します。ここでの *N* は、エラーの行とコラムを指します。

semantic error: no match for probe point, while resolving probe point pnt

イベント/ハンドラー関数 **pnt** は、さまざまな理由で解決できませんでした。このエラーは、スクリプトにイベント **kernel.function("name")** が含まれ、*name* が存在しない場合に発生します。このエラーはスクリプトに無効なカーネルファイル名やソース行数が含まれていることを意味する場合もあります。

semantic error: unresolved target-symbol expression

スクリプトのハンドラーはターゲット変数を参照しますが、変数の値が解決できませんでした。このエラーは、ハンドラーがターゲット変数を参照した際にその変数がコンテキスト内で無効であることも意味します。これは、生成されたコードのコンパイラ最適化の結果である可能性もあります。

semantic error: libdwfl failure

デバッグ情報の処理に問題がありました。ほとんどの場合、このエラーは kernel-debuginfo パッケージのインストールが原因で発生します。インストールされた kernel-debuginfo パッケージ自体に、一貫性または正確性の問題がある可能性があります。

semantic error: cannot find package debuginfo

SystemTap は適切な kernel-debuginfo を見つけることができませんでした。

5.2. ランタイムエラーおよび警告

ランタイムエラーと警告は、SystemTap インストールメンテーションがインストールされ、システム上でデータを収集すると発生します。

WARNING: Number of errors: *N*, skipped probes: *M*

この実行中に、エラー、プローブの省略、またはその両方が発生しました。*N* と *M* は、発生したエラーの数と、特定の時間間隔でイベントハンドラーを実行するのに時間がかかりすぎるなどの条件によりスキップされたプローブの数です。

division by 0

スクリプトコードが無効な除算を実行しました。

aggregate element not found

値が累積されていない集合で、**@count** 以外の統計情報抽出関数が呼び出されました。これは、division by zero と類似したものです。

aggregation overflow

現時点で、集計値を含むアレイに明確なキーのペアが過剰にあります。

MAXNESTING exceeded

関数呼び出しの入れ子の試行が多すぎます。許可されるデフォルトの関数呼び出しの入れ子は 10 です。

MAXACTION exceeded

プローブハンドラーによるステートメントの実行の試みが多すぎました。プローブハンドラー内で許可されるデフォルトのアクション数は 1000 です。

kernel/user string copy fault at *ADDR*

プローブハンドラーが、無効なアドレス (*ADDR*) でカーネルまたはユーザースペースから文字列をコピーしようとしていました。

pointer dereference fault

ターゲット変数の評価など、ポインターの逆参照演算中に障害が発生しました。

第6章 リファレンス

本章では、SystemTap についてさらに詳細な記載をしている他の参考文献を紹介しています。高度なプローブや tapset を記述する際の参考にしてください。

SystemTap Wiki

『SystemTap Wiki』とは、SystemTap のデプロイメント、使用、および開発に関連するリンクや記事がまとめられているページです。<http://sourceware.org/systemtap/wiki/HomePage> をご覧ください。

SystemTap チュートリアル

本ガイドのコンテンツの多くは『SystemTap Tutorial』を元としています。『SystemTap Tutorial』は、C++ やカーネル開発などに関して中から高程度の知識を有するユーザー向けの内容になります。<http://sourceware.org/systemtap/tutorial/> をご覧ください。

man stapprobes

man ページの **stapprobes** では、SystemTap で対応している各種のプローブポイントおよび SystemTap tapset ライブラリーで定義している他のエイリアスを紹介しています。man ページの下部には、**stapprobes.scsi**、**stapprobes.kprocess**、**stapprobes.signal** など、特定のシステムコンポーネントの同様のプローブポイントを列挙する他の man ページのリストが含まれています。

man stapfuncs

man ページの **stapfuncs** は、SystemTap tapset ライブラリーで対応している多数の関数および各関数に規定の構文を説明しています。ただし、対応している関数が **すべて** 記載されているわけではない点に注意してください。説明がない関数が他にもあります。

SystemTap Language Reference (SystemTap 言語リファレンス)

本ガイドは、SystemTap の言語設定および構文がすべて記載されているリファレンスです。C++ および他の似たようなプログラミング言語の基本的な知識または中程度の知識をお持ちのユーザーを対象としています。『SystemTap Language Reference』は、すべてのユーザーを対象に <http://sourceware.org/systemtap/langref/> で公開されています。

Tapset Developers Guide

SystemTap スクリプトの記述に熟達したら、独自のタップセットの記述に挑戦することができます。『Tapset Developers Guide』では、タップセットライブラリーに関数を追加する方法を説明しています。

Test Suite

systemtap-testsuite パッケージを使用すると、ソースからビルドしなくても SystemTap ツールチェーン全体をテストできます。また、研究や検証対象となる SystemTap スクリプトのサンプルも数多く収納されています。一部のサンプルスクリプトは、[4章 便利な SystemTap スクリプト](#) で説明しています。

デフォルトでは、**systemtap-testsuite** に含まれるサンプルスクリプトは `/usr/share/systemtap/testsuite/systemtap.examples` にあります。

付録A 更新履歴

改訂 7-7 Release for Red Hat Enterprise Linux 7.7 GA.	Mon Aug 5 2019	Vladimír Slávik
改訂 7-6 Release for Red Hat Enterprise Linux 7.6 GA.	Tue Oct 30 2018	Vladimír Slávik
改訂 7-5.1 Use an id for the stap test screen.	Mon Jun 18 2018	Radek Bíba
改訂 7-5 Release for Red Hat Enterprise Linux 7.5 Beta.	Tue Jan 09 2018	Vladimír Slávik
改訂 7-4 Release for Red Hat Enterprise Linux 7.4.	Wed Jul 26 2017	Vladimír Slávik
改訂 7-3.9 7.4 ベータ版のビルド	Tue May 16 2017	Robert Krátký
改訂 1-8 Release for Red Hat Enterprise Linux 7.3.	Wed Oct 19 2016	Robert Kratky
改訂 1-6 Async release with many fixes.	Wed Jan 20 2016	Robert Kratky
改訂 1-5 Release for Red Hat Enterprise Linux 7.2.	Thu Nov 11 2015	Robert Kratky
改訂 0-3 Release for Red Hat Enterprise Linux 7.0.	Fri Dec 6 2013	Jacquelynn East

索引

シンボル

\$count

サンプル使用例

ローカル変数, [ファイル読み取り/書き込みの I/O 時間の追跡](#)

\$return

サンプル使用例

ローカル変数, [ディスク読み取り/書き込みトラフィックの要約, 累積 I/O の追跡](#)

'foreach' 中に変数が修正される

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

@avg (整数抽出)

統計集計の計算

アレイ演算, [統計集計の計算](#)

@count (整数抽出)

統計集計の計算

アレイ演算, [統計集計の計算](#)

@max (整数抽出)

統計集計の計算

アレイ演算, [統計集計の計算](#)

@min (整数抽出)

統計集計の計算

アレイ演算, [統計集計の計算](#)

@sum (整数抽出)

統計集計の計算

アレイ演算, [統計集計の計算](#)

はじめに

SystemTap の機能, [SystemTap の機能](#)

パフォーマンスの監視, [はじめに](#)

本ガイドの目的, [本ガイドの目的](#)

目的、ドキュメント, [本ガイドの目的](#)

アグリゲーションのオーバーフロー

ランタイムエラー/警告

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

アレイ, [連想アレイ](#)

(参照 [連想アレイ](#))

アレイ/アレイ要素の消去

アレイ演算, [アレイおよびアレイ要素の消去/削除](#)

Delete 演算子, [アレイおよびアレイ要素の消去/削除](#)

仮想ファイルシステムの読み込み (累積的以外)、集計, [アレイおよびアレイ要素の消去/削除](#)

同一プローブ内の複数のアレイ演算, [アレイおよびアレイ要素の消去/削除](#)

アレイからの値の読み取り

アレイ演算, [アレイからの値の読み取り](#)

シンプルな計算でのアレイの使用, [アレイからの値の読み取り](#)

空の一意のキー, [アレイからの値の読み取り](#)

タイムスタンプ差分の計算

アレイ演算, [アレイからの値の読み取り](#)

アレイを使った代数公式

アレイからの値の読み取り

アレイ演算, [アレイからの値の読み取り](#)

アレイメンバーシップのテスト

条件付きステートメント、アレイの使用

アレイ演算, [条件付きステートメントにおけるアレイの使用](#)

アレイローカルがサポートされない

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

アレイ内での複数要素の処理

foreach

アレイ演算, [アレイ内での複数要素の処理](#)

foreach 出力の制限

アレイ演算, [アレイ内での複数要素の処理](#)

foreach 出力の順序

アレイ演算, [アレイ内での複数要素の処理](#)

アレイ演算, [アレイ内での複数要素の処理](#)

累積的仮想ファイルシステムの読み込み、集計

アレイ演算, [アレイ内での複数要素の処理](#)

アレイ内の複数要素

アレイ演算, [アレイ内での複数要素の処理](#)

アレイ演算

アレイ/アレイ要素の消去, [アレイおよびアレイ要素の消去/削除](#)

Delete 演算子, [アレイおよびアレイ要素の消去/削除](#)

仮想ファイルシステムの読み込み (累積的以外)、集計、[アレイおよびアレイ要素の消去/削除](#)

同一プローブ内の複数のアレイ演算、[アレイおよびアレイ要素の消去/削除](#)

[アレイおよびアレイ要素の削除](#)、[アレイおよびアレイ要素の消去/削除](#)

[アレイからの値の読み取り](#)、[アレイからの値の読み取り](#)

シンプルな計算でのアレイの使用、[アレイからの値の読み取り](#)

タイムスタンプ差分の計算、[アレイからの値の読み取り](#)

空の一意のキー、[アレイからの値の読み取り](#)

[アレイ内での複数要素の処理](#)、[アレイ内での複数要素の処理](#)

foreach、[アレイ内での複数要素の処理](#)

foreach 出力の制限、[アレイ内での複数要素の処理](#)

foreach 出力の順序、[アレイ内での複数要素の処理](#)

反復、アレイ内の要素の処理、[アレイ内での複数要素の処理](#)

累積的仮想ファイルシステムの読み込み、集計、[アレイ内での複数要素の処理](#)

[アレイ内の複数要素](#)、[アレイ内での複数要素の処理](#)

条件付きステートメント、アレイの使用、[条件付きステートメントにおけるアレイの使用](#)

[アレイメンバーシップのテスト](#)、[条件付きステートメントにおけるアレイの使用](#)

[統計集計の計算](#)、[統計集計の計算](#)

@avg (整数抽出)、[統計集計の計算](#)

@count (整数抽出)、[統計集計の計算](#)

@max (整数抽出)、[統計集計の計算](#)

@min (整数抽出)、[統計集計の計算](#)

@sum (整数抽出)、[統計集計の計算](#)

count (演算子)、[統計集計の計算](#)

[統計集計が収集したデータの抽出](#)、[統計集計の計算](#)

[統計集計への値の追加](#)、[統計集計の計算](#)

[連想アレイ](#)、[SystemTap でのアレイ演算](#)

[関連する値の割り当て](#)、[関連する値の割り当て](#)

[タイムスタンプをプロセス名に関連付ける](#)、[関連する値の割り当て](#)

[関連する値の増加](#)、[関連する値の増加](#)

[仮想ファイルシステムの読み込みの集計 \(VFS 読み込み\)](#)、[関連する値の増加](#)

アーキテクチャー表記法、決定、[必要なカーネル情報パッケージのインストール](#)

アーキテクチャー表記法の決定, [必要なカーネル情報パッケージのインストール](#)

イベント

[begin, イベント](#)

[end, イベント](#)

[kernel.function\("function"\), イベント](#)

[kernel.trace\("tracepoint"\), イベント](#)

[module\("module"\), イベント](#)

[syscall.system_call, イベント](#)

[timer イベント, イベント](#)

[vfs.file_operation, イベント](#)

[ワイルドカード, イベント](#)

[同期および非同期イベントの例, イベント](#)

[同期イベント, イベント](#)

[概要, イベント](#)

[非同期イベント, イベント](#)

[イベントおよびハンドラー, SystemTap の作動方法](#)

イベントタイプ

[SystemTap の作動方法, SystemTap の作動方法](#)

[イベントワイルドカード, イベント](#)

[イベント内のワイルドカード, イベント](#)

インストルメンテーションモジュール

[クロスインストルメンテーション, 他のコンピューター用のインストルメンテーション生成](#)

インストール

[systemtap パッケージ, SystemTap のインストール](#)

[systemtap-runtime パッケージ, SystemTap のインストール](#)

[カーネルバージョン、判断, 必要なカーネル情報パッケージのインストール](#)

[カーネル情報パッケージ, 必要なカーネル情報パッケージのインストール](#)

[初期テスト, 初期テスト](#)

[必要なパッケージ, 必要なカーネル情報パッケージのインストール](#)

[設定とインストール, インストールと設定](#)

インデックス式

[概要](#)

アレイ, [連想アレイ](#)

エラー

ランタイムエラー/警告, [ランタイムエラーおよび警告](#)
 aggregate element not found, [ランタイムエラーおよび警告](#)
 aggregation overflow, [ランタイムエラーおよび警告](#)
 division by 0, [ランタイムエラーおよび警告](#)
 MAXACTION exceeded, [ランタイムエラーおよび警告](#)
 MAXNESTING exceeded, [ランタイムエラーおよび警告](#)
 number of errors: N, skipped probes: M, [ランタイムエラーおよび警告](#)
 pointer dereference fault, [ランタイムエラーおよび警告](#)
 コピー障害, [ランタイムエラーおよび警告](#)

[解析エラー/セマンティックエラー](#), [解析エラーとセマンティックエラー](#)
 'foreach' 中に変数が修正される, [解析エラーとセマンティックエラー](#)
 guru モード, [解析エラーとセマンティックエラー](#)
 libdwfl 失敗, [解析エラーとセマンティックエラー](#)
 スクリプトの文法的なエラー/誤字のエラー, [解析エラーとセマンティックエラー](#)
 プローブの不一致, [解析エラーとセマンティックエラー](#)
 プローブポイントに一致するものがない, [解析エラーとセマンティックエラー](#)
 変数/アレイに無効な値, [解析エラーとセマンティックエラー](#)
 特権のないスクリプトにコードが埋め込まれている, [解析エラーとセマンティックエラー](#)
 解決できないターゲット記号式, [解析エラーとセマンティックエラー](#)
 解決できない識別子のタイプ, [解析エラーとセマンティックエラー](#)
 解決できない関数呼び出し, [解析エラーとセマンティックエラー](#)
 記号やアレイに期待されているインデックス式, [解析エラーとセマンティックエラー](#)
 識別子タイプの不一致, [解析エラーとセマンティックエラー](#)
 非グローバルアレイ, [解析エラーとセマンティックエラー](#)

オプション、stap

用途, [SystemTap スクリプトの実行](#)

カーネル/ユーザースペースで費やした時間、判定

[SystemTap スクリプトの例](#), [カーネルおよびユーザースペースで費やした時間の判定](#)

カーネルおよびユーザースペース、費やした時間の判定

SystemTap スクリプトの例, [カーネルおよびユーザースペースで費やした時間の判定](#)

カーネルおよびユーザースペースで費やした時間の判定

SystemTap スクリプトの例, [カーネルおよびユーザースペースで費やした時間の判定](#)

カーネルバージョン、判断, [必要なカーネル情報パッケージのインストール](#)

カーネルバージョンの判断, [必要なカーネル情報パッケージのインストール](#)

カーネル情報パッケージ, [必要なカーネル情報パッケージのインストール](#)

キーのペア

概要

[アレイ](#), [連想アレイ](#)

クロスインストールメンターション

SystemTap スクリプトからインストールメンターションを生成する, [他のコンピューター用のインストールメンターション生成](#)

SystemTap スクリプトからカーネルモジュールを構築する, [他のコンピューター用のインストールメンターション生成](#)

インストールメンターションモジュール, [他のコンピューター用のインストールメンターション生成](#)

ターゲットカーネル, [他のコンピューター用のインストールメンターション生成](#)

ターゲットシステム, [他のコンピューター用のインストールメンターション生成](#)

ホストシステム, [他のコンピューター用のインストールメンターション生成](#)

利点, [他のコンピューター用のインストールメンターション生成](#)

設定

ホストシステムおよびターゲットシステム, [他のコンピューター用のインストールメンターション生成](#)

クロスインストールメンターションの利点, [他のコンピューター用のインストールメンターション生成](#)

コピー障害

[ランタイムエラー/警告](#)

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

コマンドラインの引数

SystemTap ハンドラーコンストラクト

[ハンドラー](#), [コマンドラインの引数](#)

コンポーネント

SystemTap スクリプト

概要, [SystemTap スクリプト](#)

システムコール、監視

SystemTap スクリプトの例, [最もよく使われるシステムコールの追跡](#)

システムコールの監視

SystemTap スクリプトの例, [最もよく使われるシステムコールの追跡](#)

システムコールの監視 (プロセスごとのボリューム)

SystemTap スクリプトの例, [プロセスごとのシステムコールボリュームの追跡](#)

システムコールボリューム (プロセスごと)、監視

SystemTap スクリプトの例, [プロセスごとのシステムコールボリュームの追跡](#)

シンプルな計算でのアレイの使用

アレイからの値の読み取り

アレイ演算, [アレイからの値の読み取り](#)

スクリプト

概要, [SystemTap スクリプト](#)

イベントおよびハンドラー, [SystemTap スクリプト](#)

コンポーネント, [SystemTap スクリプト](#)

形式および構文, [SystemTap スクリプト](#)

関数, [SystemTap スクリプト](#)

スクリプトの例

CPU ティック, [カーネルおよびユーザースペースで費やした時間の判定](#)

ctime(), 使用例, [ディスク読み取り/書き込みトラフィックの要約](#)

futex システムコール, [競合ユーザースペースのロックの特定](#)

I/O 時間の監視, [ファイル読み取り/書き込みの I/O 時間の追跡](#)

if/else 条件、代替の構文, [ネットワークのプロファイリング](#)

inode 番号, [ファイルの読み取りおよび書き込みの監視](#)

net/socket.c、関数の追跡, [ネットワークソケットコードで呼び出された関数の追跡](#)

stat -c、ファイルデバイス番号の決定 (整数形式), [ファイルの読み取りおよび書き込みの監視](#)

stat -c、全体デバイス番号の決定, [I/O 監視 \(デバイスごと\)](#)

thread_indent(), サンプル使用例, [呼び出し先の追跡](#)

timer.ms(), サンプル使用例, [関数呼び出しのカウンタ](#)

timer.s(), サンプル使用例, [ポーリングアプリケーションの監視, 最もよく使われるシステムコールの追跡](#)

trigger function, [呼び出し先の追跡](#)

[usrdev2kerndev\(\)](#), [I/O 監視 \(デバイスごと\)](#)

[カーネルおよびユーザースペースで費やした時間の判定](#), [カーネルおよびユーザースペースで費やした時間の判定](#)

[システムコールの監視](#), [最もよく使われるシステムコールの追跡](#)

[システムコールの監視 \(プロセスごとのボリューム\)](#), [プロセスごとのシステムコールボリュームの追跡](#)

[ディスク I/O トラフィックの要約](#), [ディスク読み取り/書き込みトラフィックの要約](#)

[デバイス I/O の監視](#), [I/O 監視 \(デバイスごと\)](#)

[ネットワークのプロファイリング](#), [ネットワークのプロファイリング](#), [カーネルでのネットワークパケットドロップの監視](#)

[ネットワークソケットコードで呼び出された関数の追跡](#), [ネットワークソケットコードで呼び出された関数の追跡](#)

[ファイルの読み取りおよび書き込みの監視](#), [ファイルの読み取りおよび書き込みの監視](#)

[ファイルデバイス番号 \(整数形式\)](#), [ファイルの読み取りおよび書き込みの監視](#)

[ファイル属性の変更の監視](#), [ファイル属性の変更の監視](#)

[プロセスのデッドロック \(futex 競合により発生\)](#), [競合ユーザースペースのロックの特定](#)

[ポーリングアプリケーションの監視](#), [ポーリングアプリケーションの監視](#)

[全体デバイス番号 \(コマンドライン引数として使用\)](#), [I/O 監視 \(デバイスごと\)](#)

[呼び出し先の追跡](#), [呼び出し先の追跡](#)

[着信 TCP 接続の監視](#), [着信 TCP 接続の監視](#)

[競合ユーザースペースのロック \(futex 競合\) の特定](#), [競合ユーザースペースのロックの特定](#)

[累積 I/O の追跡](#), [累積 I/O の追跡](#)

[複数のコマンドライン引数、例](#), [呼び出し先の追跡](#)

[関数呼び出しの集計](#), [関数呼び出しのカウント](#)

スクリプトの文法的なエラー/誤字のエラー

[解析エラー/セマンティックエラー](#)

[SystemTap のエラーを理解する](#), [解析エラーとセマンティックエラー](#)

スクリプトの誤字のエラー

[解析エラー/セマンティックエラー](#)

[SystemTap のエラーを理解する](#), [解析エラーとセマンティックエラー](#)

ステートメントブロック

[SystemTap スクリプト](#)

[概要](#), [SystemTap スクリプト](#)

セッション、SystemTap, [アーキテクチャー](#)

タイムスタンプ、プロセス名との関連付け
関連する値の割り当て
アレイ演算, [関連する値の割り当て](#)

タイムスタンプをプロセス名に関連付ける
関連する値の割り当て
アレイ演算, [関連する値の割り当て](#)

タイムスタンプ差分、計算
アレイからの値の読み取り
アレイ演算, [アレイからの値の読み取り](#)

タイムスタンプ差分の計算
アレイからの値の読み取り
アレイ演算, [アレイからの値の読み取り](#)

ターゲットカーネル
[クロスインストールメンテーション](#), [他のコンピューター用のインストールメンテーション生成](#)

ターゲットシステム
[クロスインストールメンテーション](#), [他のコンピューター用のインストールメンテーション生成](#)

ターゲットシステムおよびホストシステム
[configuration](#), [他のコンピューター用のインストールメンテーション生成](#)

ターゲット記号式、解決できない
[解析エラー/セマンティックエラー](#)
SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

テスト、初期, [初期テスト](#)

ディスク I/O トラフィック、要約
スクリプトの例, [ディスク読み取り/書き込みトラフィックの要約](#)

ディスク I/O トラフィックの要約
スクリプトの例, [ディスク読み取り/書き込みトラフィックの要約](#)

デバイス I/O の監視
SystemTap スクリプトの例, [I/O 監視 \(デバイスごと\)](#)

デバイス I/O、監視

SystemTap スクリプトの例, [I/O 監視 \(デバイスごと\)](#)

ネットワークのプロファイリング

SystemTap スクリプトの例, [ネットワークのプロファイリング](#), [カーネルでのネットワークパケットドロップの監視](#)

ネットワークソケットコード、呼び出された関数の追跡

SystemTap スクリプトの例, [ネットワークソケットコードで呼び出された関数の追跡](#)

ネットワークソケットコードで呼び出された関数、追跡

SystemTap スクリプトの例, [ネットワークソケットコードで呼び出された関数の追跡](#)

ネットワークソケットコードで呼び出された関数の追跡

SystemTap スクリプトの例, [ネットワークソケットコードで呼び出された関数の追跡](#)

ネットワークトラフィック、監視

SystemTap スクリプトの例, [ネットワークのプロファイリング](#), [カーネルでのネットワークパケットドロップの監視](#)

ハンドラー

SystemTap ハンドラーコンストラクト, [基本的な SystemTap ハンドラーコンストラクト](#)
コマンドラインの引数, [コマンドラインの引数](#)

変数, [変数](#)

構文および形式, [基本的な SystemTap ハンドラーコンストラクト](#)

条件付き (conditional) ステートメント, [条件付き \(conditional\) ステートメント](#)

For ループ, [条件付き \(conditional\) ステートメント](#)

if/else, [条件付き \(conditional\) ステートメント](#)

While ループ, [条件付き \(conditional\) ステートメント](#)

条件演算子, [条件付き \(conditional\) ステートメント](#)

概要, [Systemtap ハンドラー/ボディー](#)

ハンドラーおよびイベント, SystemTap の作動方法

SystemTap スクリプト

概要, [SystemTap スクリプト](#)

ハンドラー関数, Systemtap ハンドラー/ボディー

パフォーマンスの監視

はじめに, [はじめに](#)

ファイルのデバイス番号 (整数形式)

SystemTap スクリプトの例, [ファイルの読み取りおよび書き込みの監視](#)

ファイルの読み取り/書き込み、監視

SystemTap スクリプトの例, [ファイルの読み取りおよび書き込みの監視](#)

ファイルの読み取りおよび書き込みの監視

SystemTap スクリプトの例, [ファイルの読み取りおよび書き込みの監視](#)

ファイルへの書き込み/読み取り、監視

SystemTap スクリプトの例, [ファイルの読み取りおよび書き込みの監視](#)

ファイルデバイス番号 (整数形式)

SystemTap スクリプトの例, [ファイルの読み取りおよび書き込みの監視](#)

ファイル属性、変更の監視

SystemTap スクリプトの例, [ファイル属性の変更の監視](#)

ファイル属性の変更、監視

SystemTap スクリプトの例, [ファイル属性の変更の監視](#)

ファイル属性の変更の監視

SystemTap スクリプトの例, [ファイル属性の変更の監視](#)

フライトレコーダーモード, SystemTap フライトレコーダーモード

ファイルモード, [ファイルフライトレコーダー](#)

メモリー内モード, [メモリー内フライトレコーダー](#)

プロセスのデッドロック (futex 競合により発生)

SystemTap スクリプトの例, [競合ユーザースペースのロックの特定](#)

プローブ

SystemTap スクリプト

概要, [SystemTap スクリプト](#)

プローブの不一致

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

プローブポイント (一致なし)

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

プローブポイントに一致するものがない

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

ホストシステム

クロスインストールメンテーション, [他のコンピューター用のインストールメンテーション生成](#)

ホストシステムおよびターゲットシステム

クロスインストールメンテーション

設定, [他のコンピューター用のインストールメンテーション生成](#)

ポーリングアプリケーション、監視

SystemTap スクリプトの例, [ポーリングアプリケーションの監視](#)

ポーリングアプリケーションの監視

SystemTap スクリプトの例, [ポーリングアプリケーションの監視](#)

メンバーシップ (アレイ内)、テスト

条件付きステートメント、アレイの使用

アレイ演算, [条件付きステートメントにおけるアレイの使用](#)

ユーザーおよびカーネルスペース、費やした時間の判定

SystemTap スクリプトの例, [カーネルおよびユーザースペースで費やした時間の判定](#)

ランタイムエラー/警告

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

aggregate element not found, [ランタイムエラーおよび警告](#)

aggregation overflow, [ランタイムエラーおよび警告](#)

division by 0, [ランタイムエラーおよび警告](#)

MAXACTION exceeded, [ランタイムエラーおよび警告](#)

MAXNESTING exceeded, [ランタイムエラーおよび警告](#)

number of errors: N, skipped probes: M, [ランタイムエラーおよび警告](#)

pointer dereference fault, [ランタイムエラーおよび警告](#)

コピー障害, [ランタイムエラーおよび警告](#)

ローカル変数

name, [Systemtap ハンドラー/ボディー](#)

サンプル使用例

\$count, [ファイル読み取り/書き込みの I/O 時間の追跡](#)

\$return, [ディスク読み取り/書き込みトラフィックの要約](#), [累積 I/O の追跡](#)

一意のキー

概要

[アレイ](#), [連想アレイ](#)

仮想ファイルシステムの読み込み (累積的)、集計

[アレイ内での複数要素の処理](#)

[アレイ演算](#), [アレイ内での複数要素の処理](#)

仮想ファイルシステムの読み込み (累積的以外)、集計

[アレイ/アレイ要素の消去](#)

[アレイ演算](#), [アレイおよびアレイ要素の消去/削除](#)

仮想ファイルシステムの読み込みの集計 (VFS 読み込み)

[関連する値の増加](#)

[アレイ演算](#), [関連する値の増加](#)

使用方法

[SystemTap スクリプトの実行](#), [SystemTap スクリプトの実行](#)

[オプション](#), [stap](#), [SystemTap スクリプトの実行](#)

[便利な SystemTap スクリプトの例](#), [便利な SystemTap スクリプト](#)

[値](#), [割り当て](#)

[アレイ演算](#), [関連する値の割り当て](#)

[全体デバイス番号 \(コマンドライン引数として使用\)](#)

[SystemTap スクリプトの例](#), [I/O 監視 \(デバイスごと\)](#)

[初期テスト](#), [初期テスト](#)

[反復](#), [アレイ内の要素の処理](#)

[アレイ内での複数要素の処理](#)

[アレイ演算](#), [アレイ内での複数要素の処理](#)

[同一プローブ内の複数のアレイ演算](#)

[アレイ/アレイ要素の消去](#)

[アレイ演算](#), [アレイおよびアレイ要素の消去/削除](#)

同期および非同期イベントの例

イベント, [イベント](#)

同期イベント

イベント, [イベント](#)

呼び出し先の追跡

SystemTap スクリプトの例, [呼び出し先の追跡](#)

変数

SystemTap ハンドラーコンストラクト

ハンドラー, [変数](#)

変数 (ローカル)

name, [Systemtap ハンドラー/ボディー](#)

サンプル使用例

\$count, [ファイル読み取り/書き込みの I/O 時間の追跡](#)

\$return, [ディスク読み取り/書き込みトラフィックの要約, 累積 I/O の追跡](#)

変数/アレイに無効な値

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

形式および構文

printf(), [Systemtap ハンドラー/ボディー](#)

SystemTap スクリプト

概要, [SystemTap スクリプト](#)

SystemTap ハンドラーコンストラクト

ハンドラー, [変数](#)

必要なパッケージ, 必要なカーネル情報パッケージのインストール

整数抽出

統計集計の計算

アレイ演算, [統計集計の計算](#)

書式指定子

printf(), [Systemtap ハンドラー/ボディー](#)

書式文字列

`printf()`, [Systemtap ハンドラー/ボディー](#)

最も大きいディスクの読み取り/書き込み、識別

スクリプトの例, [ディスク読み取り/書き込みトラフィックの要約](#)

最も大きいディスクの読み取り/書き込みの特定

スクリプトの例, [ディスク読み取り/書き込みトラフィックの要約](#)

本ガイドの目的

はじめに, [本ガイドの目的](#)

条件付きステートメント、アレイの使用

アレイ演算, [条件付きステートメントにおけるアレイの使用](#)

アレイメンバーシップのテスト, [条件付きステートメントにおけるアレイの使用](#)

条件演算子

条件付き (conditional) ステートメント

handlers, [条件付き \(conditional\) ステートメント](#)

構文

概要

アレイ, [連想アレイ](#)

構文および形式

`printf()`, [Systemtap ハンドラー/ボディー](#)

SystemTap スクリプト

概要, [SystemTap スクリプト](#)

SystemTap ハンドラーコンストラクト

ハンドラー, [基本的な SystemTap ハンドラーコンストラクト](#)

標準入力から、スクリプトを実行

用途, [SystemTap スクリプトの実行](#)

標準入力からスクリプトを実行, SystemTap スクリプトの実行

無効な除算

ランタイムエラー/警告

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

特権のないスクリプト、コードが埋め込まれている

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

特権のないスクリプトにコードが埋め込まれている

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

特権のないスクリプトに安全でないコードが埋め込まれている

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

用途

stap, [SystemTap スクリプトの実行](#)

stapdev, [SystemTap スクリプトの実行](#)

staprun, [SystemTap スクリプトの実行](#)

stapusr, [SystemTap スクリプトの実行](#)

標準入力から、スクリプトを実行, [SystemTap スクリプトの実行](#)

目的、ドキュメント

はじめに, [本ガイドの目的](#)

着信 TCP 接続、監視

SystemTap スクリプトの例, [着信 TCP 接続の監視](#)

着信 TCP 接続の監視

SystemTap スクリプトの例, [着信 TCP 接続の監視](#)

着信/発信の関数呼び出し、追跡

SystemTap スクリプトの例, [呼び出し先の追跡](#)

着信/発信の関数呼び出しの追跡

SystemTap スクリプトの例, [呼び出し先の追跡](#)

空の一意のキー

アレイからの値の読み取り

アレイ演算, [アレイからの値の読み取り](#)

競合ユーザースペースのロック (futex 競合) の特定

SystemTap スクリプトの例, [競合ユーザースペースのロックの特定](#)

競合ユーザースペースのロック (futex 競合)、特定

SystemTap スクリプトの例, [競合ユーザースペースのロックの特定](#)

累積 I/O の監視

SystemTap スクリプトの例, [累積 I/O の追跡](#)

累積 I/O の追跡

SystemTap スクリプトの例, [累積 I/O の追跡](#)

累積 I/O、追跡

SystemTap スクリプトの例, [累積 I/O の追跡](#)

累積的仮想ファイルシステムの読み込み、集計

アレイ内での複数要素の処理

アレイ演算, [アレイ内での複数要素の処理](#)

統計集計

アレイ演算, [統計集計の計算](#)

統計集計が収集したデータの抽出

統計集計の計算

アレイ演算, [統計集計の計算](#)

統計集計の計算

アレイ演算, [統計集計の計算](#)

@avg (整数抽出), [統計集計の計算](#)

@count (整数抽出), [統計集計の計算](#)

@max (整数抽出), [統計集計の計算](#)

@min (整数抽出), [統計集計の計算](#)

@sum (整数抽出), [統計集計の計算](#)

count (演算子), [統計集計の計算](#)

統計集計が収集したデータの抽出, [統計集計の計算](#)

統計集計への値の追加, [統計集計の計算](#)

統計集計への値の追加

統計集計の計算

アレイ演算, [統計集計の計算](#)

複数のコマンドライン引数、例

SystemTap スクリプトの例, [呼び出し先の追跡](#)

複数のコマンドライン引数の例

SystemTap スクリプトの例, [呼び出し先の追跡](#)

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

'foreach' 中に変数が修正される, [解析エラーとセマンティックエラー](#)

guru モード, [解析エラーとセマンティックエラー](#)

libdwfl 失敗, [解析エラーとセマンティックエラー](#)

スクリプトの文法的なエラー/誤字のエラー, [解析エラーとセマンティックエラー](#)

プローブの不一致, [解析エラーとセマンティックエラー](#)

プローブポイントに一致するものがない, [解析エラーとセマンティックエラー](#)

変数/アレイに無効な値, [解析エラーとセマンティックエラー](#)

特権のないスクリプトにコードが埋め込まれている, [解析エラーとセマンティックエラー](#)

解決できないターゲット記号式, [解析エラーとセマンティックエラー](#)

解決できない識別子のタイプ, [解析エラーとセマンティックエラー](#)

解決できない関数呼び出し, [解析エラーとセマンティックエラー](#)

記号やアレイに期待されているインデックス式, [解析エラーとセマンティックエラー](#)

識別子タイプの不一致, [解析エラーとセマンティックエラー](#)

非グローバルアレイ, [解析エラーとセマンティックエラー](#)

解決できないターゲット記号式

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

解決できない識別子のタイプ

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

解決できない関数呼び出し

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

記号やアレイに期待されているインデックス式

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

設定とインストール, インストールと設定

識別子タイプの不一致

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

連想アレイ

概要, [連想アレイ](#)

example, [連想アレイ](#)

インデックス式, [連想アレイ](#)

キーのペア, [連想アレイ](#)

一意のキー, [連想アレイ](#)

構文, [連想アレイ](#)

関連する値, [連想アレイ](#)

関数, [Systemtap ハンドラー/ボディー](#)

cpu(), [Systemtap ハンドラー/ボディー](#)

ctime(), [Systemtap ハンドラー/ボディー](#)

gettimeofday_s(), [Systemtap ハンドラー/ボディー](#)

pp(), [Systemtap ハンドラー/ボディー](#)

SystemTap スクリプト

概要, [SystemTap スクリプト](#)

target(), [Systemtap ハンドラー/ボディー](#)

thread_indent(), [Systemtap ハンドラー/ボディー](#)

tid(), [Systemtap ハンドラー/ボディー](#)

uid(), [Systemtap ハンドラー/ボディー](#)

関数 (ハンドラーで使用)

exit(), [Systemtap ハンドラー/ボディー](#)

関数呼び出し (着信/発信)、追跡

SystemTap スクリプトの例, [呼び出し先の追跡](#)

関数呼び出し (解決できない)

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

関数呼び出し、集計

SystemTap スクリプトの例, [関数呼び出しのカウン](#)

関数呼び出しのカウン

SystemTap スクリプトの例, [関数呼び出しのカウン](#)

関数呼び出しの集計

SystemTap スクリプトの例, [関数呼び出しのカウン](#)

関連する値

概要

[アレイ](#), [連想アレイ](#)

関連する値の割り当て

[アレイ演算](#), [関連する値の割り当て](#)

[タイムスタンプをプロセス名に関連付ける](#), [関連する値の割り当て](#)

[タイムスタンプをプロセス名に関連付ける](#)

[アレイ演算](#), [関連する値の割り当て](#)

関連する値の増加

[アレイ演算](#), [関連する値の増加](#)

[仮想ファイルシステムの読み込みの集計 \(VFS 読み込み\)](#), [関連する値の増加](#)

集計 (統計)

[アレイ演算](#), [統計集計の計算](#)

非グローバルアレイ

[解析エラー/セマンティックエラー](#)

[SystemTap のエラーを理解する](#), [解析エラーとセマンティックエラー](#)

非同期イベント

[イベント](#), [イベント](#)

A

aggregate element not found

[ランタイムエラー/警告](#)

[SystemTap のエラーを理解する](#), [ランタイムエラーおよび警告](#)

aggregation overflow

ランタイムエラー/警告

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

B

begin

イベント, [イベント](#)

C

CONFIG_HZ, 計算, [変数](#)

count 演算子

統計集計の計算

array (演算子), [統計集計の計算](#)

CPU ティック

SystemTap スクリプトの例, [カーネルおよびユーザースペースで費やした時間の判定](#)

cpu()

関数, [Systemtap ハンドラー/ボディー](#)

cross-compiling, [他のコンピューター用のインストルメンテーション生成](#)

ctime()

関数, [Systemtap ハンドラー/ボディー](#)

ctime(), 使用例

スクリプトの例, [ディスク読み取り/書き込みトラフィックの要約](#)

D

Delete 演算子

アレイ/アレイ要素の消去

アレイ演算, [アレイおよびアレイ要素の消去/削除](#)

division by 0

ランタイムエラー/警告

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

E

end

イベント, [イベント](#)

example

概要

アレイ, [連想アレイ](#)

exceeded MAXACTION

ランタイムエラー/警告

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

exceeded MAXNESTING

ランタイムエラー/警告

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

exit()

関数, [Systemtap ハンドラー/ボディー](#)

F

For ループ

条件付き (conditional) ステートメント

ハンドラー, [条件付き \(conditional\) ステートメント](#)

foreach

アレイ内での複数要素の処理

アレイ演算, [アレイ内での複数要素の処理](#)

foreach 出力の制限

アレイ内での複数要素の処理

アレイ演算, [アレイ内での複数要素の処理](#)

foreach 出力の順序

アレイ内での複数要素の処理

アレイ演算, [アレイ内での複数要素の処理](#)

format

概要

アレイ, [連想アレイ](#)

futex システムコール

SystemTap スクリプトの例, [競合ユーザースペースのロックの特定](#)

futex 競合、定義

SystemTap スクリプトの例, [競合ユーザースペースのロックの特定](#)

futex 競合、特定

SystemTap スクリプトの例, [競合ユーザースペースのロックの特定](#)

G

gettimeofday_s()

関数, [Systemtap ハンドラー/ボディー](#)

global

SystemTap ハンドラーコンストラクト
ハンドラー, [変数](#)

guru モード

解析エラー/セマンティックエラー
SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

H

handlers

SystemTap ハンドラーコンストラクト
global, [変数](#)

I

I/O の時間

SystemTap スクリプトの例, [ファイル読み取り/書き込みの I/O 時間の追跡](#)

I/O アクティビティの出力 (累積)

SystemTap スクリプトの例, [累積 I/O の追跡](#)

I/O トラフィック、要約

スクリプトの例, [ディスク読み取り/書き込みトラフィックの要約](#)

I/O 時間、監視

SystemTap スクリプトの例, [ファイル読み取り/書き込みの I/O 時間の追跡](#)

I/O 時間の監視

SystemTap スクリプトの例, [ファイル読み取り/書き込みの I/O 時間の追跡](#)

I/O 監視 (デバイスごと)

SystemTap スクリプトの例, [I/O 監視 \(デバイスごと\)](#)

if/else

条件付き (conditional) ステートメント

ハンドラー, [条件付き \(conditional\) ステートメント](#)

if/else ステートメント、アレイの使用

アレイ演算, [条件付きステートメントにおけるアレイの使用](#)

if/else 条件、代替の構文

SystemTap スクリプトの例, [ネットワークのプロファイリング](#)

inode 番号

SystemTap スクリプトの例, [ファイルの読み取りおよび書き込みの監視](#)

K

kernel.function("function")

イベント, [イベント](#)

kernel.trace("tracepoint")

イベント, [イベント](#)

L

libdwfl 失敗

解析エラー/セマンティックエラー

SystemTap のエラーを理解する, [解析エラーとセマンティックエラー](#)

M

MAXACTION exceeded

ランタイムエラー/警告

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

MAXNESTING exceeded

ランタイムエラー/警告

SystemTap のエラーを理解する, [ランタイムエラーおよび警告](#)

module("module")
 イベント, イベント

N

name
 ローカル変数, Systemtap ハンドラー/ボディー

net/socket.c、関数の追跡
 SystemTap スクリプトの例, ネットワークソケットコードで呼び出された関数の追跡

number of errors: N, skipped probes: M
 ランタイムエラー/警告
 SystemTap のエラーを理解する, ランタイムエラーおよび警告

O

operations
 アレイ/アレイ要素の消去, アレイおよびアレイ要素の消去/削除
 Delete 演算子, アレイおよびアレイ要素の消去/削除
 仮想ファイルシステムの読み込み (累積的以外)、集計, アレイおよびアレイ要素の消去/削除
 同一プローブ内の複数のアレイ演算, アレイおよびアレイ要素の消去/削除

 アレイおよびアレイ要素の削除, アレイおよびアレイ要素の消去/削除
 アレイからの値の読み取り, アレイからの値の読み取り
 シンプルな計算でのアレイの使用, アレイからの値の読み取り
 タイムスタンプ差分の計算, アレイからの値の読み取り
 空の一意のキー, アレイからの値の読み取り

 アレイ内での複数要素の処理, アレイ内での複数要素の処理
 foreach, アレイ内での複数要素の処理
 foreach 出力の制限, アレイ内での複数要素の処理
 foreach 出力の順序, アレイ内での複数要素の処理
 反復、アレイ内の要素の処理, アレイ内での複数要素の処理
 累積的仮想ファイルシステムの読み込み、集計, アレイ内での複数要素の処理

 アレイ内の複数要素, アレイ内での複数要素の処理
 条件付きステートメント、アレイの使用, 条件付きステートメントにおけるアレイの使用
 アレイメンバーシップのテスト, 条件付きステートメントにおけるアレイの使用

統計集計の計算, [統計集計の計算](#)

[@avg \(整数抽出\)](#), [統計集計の計算](#)

[@count \(整数抽出\)](#), [統計集計の計算](#)

[@max \(整数抽出\)](#), [統計集計の計算](#)

[@min \(整数抽出\)](#), [統計集計の計算](#)

[@sum \(整数抽出\)](#), [統計集計の計算](#)

[count \(演算子\)](#), [統計集計の計算](#)

[統計集計が収集したデータの抽出](#), [統計集計の計算](#)

[統計集計への値の追加](#), [統計集計の計算](#)

連想アレイ, [SystemTap でのアレイ演算](#)

関連する値の割り当て

[タイムスタンプをプロセス名に関連付ける](#), [関連する値の割り当て](#)

関連する値の増加, [関連する値の増加](#)

[仮想ファイルシステムの読み込みの集計 \(VFS 読み込み\)](#), [関連する値の増加](#)

P

pointer dereference fault

[ランタイムエラー/警告](#)

[SystemTap のエラーを理解する](#), [ランタイムエラーおよび警告](#)

pp()

[関数](#), [Systemtap ハンドラー/ボディー](#)

printf()

[書式指定子](#), [Systemtap ハンドラー/ボディー](#)

[書式文字列](#), [Systemtap ハンドラー/ボディー](#)

[構文および形式](#), [Systemtap ハンドラー/ボディー](#)

S

scripts

[概要](#)

[ステートメントブロック](#), [SystemTap スクリプト](#)

[プローブ](#), [SystemTap スクリプト](#)

stap

用途, [SystemTap スクリプトの実行](#)

stap オプション, [SystemTap スクリプトの実行](#)

stapdev

使用方法, [SystemTap スクリプトの実行](#)

staprun

使用方法, [SystemTap スクリプトの実行](#)

stapusr

使用方法, [SystemTap スクリプトの実行](#)

stat -c、ファイルデバイス番号の決定 (整数形式)

[SystemTap スクリプトの例](#), [ファイルの読み取りおよび書き込みの監視](#)

stat -c、全体デバイス番号の決定

[SystemTap スクリプトの例](#), [I/O 監視 \(デバイスごと\)](#)

syscall.system_call

[イベント](#), [イベント](#)

[SystemTap のアーキテクチャー](#), [アーキテクチャー](#)

[SystemTap のエラーを理解する](#)

[ランタイムエラー/警告](#), [ランタイムエラーおよび警告](#)

[aggregate element not found](#), [ランタイムエラーおよび警告](#)

[aggregation overflow](#), [ランタイムエラーおよび警告](#)

[division by 0](#), [ランタイムエラーおよび警告](#)

[MAXACTION exceeded](#), [ランタイムエラーおよび警告](#)

[MAXNESTING exceeded](#), [ランタイムエラーおよび警告](#)

[number of errors: N, skipped probes: M](#), [ランタイムエラーおよび警告](#)

[pointer dereference fault](#), [ランタイムエラーおよび警告](#)

[コピー障害](#), [ランタイムエラーおよび警告](#)

[解析エラー/セマンティックエラー](#), [解析エラーとセマンティックエラー](#)

['foreach' 中に変数が修正される](#), [解析エラーとセマンティックエラー](#)

[guru モード](#), [解析エラーとセマンティックエラー](#)

[libdwfl 失敗](#), [解析エラーとセマンティックエラー](#)

[スクリプトの文法的なエラー/誤字のエラー](#), [解析エラーとセマンティックエラー](#)

プローブの不一致, [解析エラーとセマンティックエラー](#)

プローブポイントに一致するものがない, [解析エラーとセマンティックエラー](#)

変数/アレイに無効な値, [解析エラーとセマンティックエラー](#)

特権のないスクリプトにコードが埋め込まれている, [解析エラーとセマンティックエラー](#)

解決できないターゲット記号式, [解析エラーとセマンティックエラー](#)

解決できない識別子のタイプ, [解析エラーとセマンティックエラー](#)

解決できない関数呼び出し, [解析エラーとセマンティックエラー](#)

記号やアレイに期待されているインデックス式, [解析エラーとセマンティックエラー](#)

識別子タイプの不一致, [解析エラーとセマンティックエラー](#)

非グローバルアレイ, [解析エラーとセマンティックエラー](#)

[SystemTap の作動方法](#), [SystemTap の作動方法](#)

[SystemTap セッション](#), [アーキテクチャー](#)

[アーキテクチャー](#), [アーキテクチャー](#)

[イベントおよびハンドラー](#), [SystemTap の作動方法](#)

[イベントタイプ](#), [SystemTap の作動方法](#)

[SystemTap の使用](#), [SystemTap の使用](#)

[SystemTap の実行に必要な RPM](#), [必要なカーネル情報パッケージのインストール](#)

[SystemTap の実行に必要なパッケージ](#), [必要なカーネル情報パッケージのインストール](#)

[SystemTap の機能](#)

[はじめに](#), [SystemTap の機能](#)

[SystemTap アーキテクチャー](#), [アーキテクチャー](#)

[SystemTap スクリプト](#)

[便利な例](#), [便利な SystemTap スクリプト](#)

[概要](#), [SystemTap スクリプト](#)

[イベントおよびハンドラー](#), [SystemTap スクリプト](#)

[コンポーネント](#), [SystemTap スクリプト](#)

[ステートメントブロック](#), [SystemTap スクリプト](#)

[プローブ](#), [SystemTap スクリプト](#)

[形式および構文](#), [SystemTap スクリプト](#)

[関数](#), [SystemTap スクリプト](#)

[SystemTap スクリプト、実行方法](#), [SystemTap スクリプトの実行](#)

SystemTap スクリプトからインストルメンテーション/カーネルモジュールをコンパイルする,他のコンピュータ用のインストルメンテーション生成

SystemTap スクリプトからインストルメンテーションモジュール、構築する,他のコンピュータ用のインストルメンテーション生成

SystemTap スクリプトからインストルメンテーションモジュールを構築する,他のコンピュータ用のインストルメンテーション生成

SystemTap スクリプトからカーネルモジュール、構築する,他のコンピュータ用のインストルメンテーション生成

SystemTap スクリプトからカーネルモジュールを構築する,他のコンピュータ用のインストルメンテーション生成

SystemTap スクリプトの例,便利な SystemTap スクリプト

CPU ティック,カーネルおよびユーザースペースで費やした時間の判定

ctime(), 使用例, ディスク読み取り/書き込みトラフィックの要約

futex システムコール, 競合ユーザースペースのロックの特定

I/O 時間の監視, ファイル読み取り/書き込みの I/O 時間の追跡

if/else 条件、代替の構文, ネットワークのプロファイリング

inode 番号, ファイルの読み取りおよび書き込みの監視

net/socket.c、関数の追跡, ネットワークソケットコードで呼び出された関数の追跡

stat -c、ファイルデバイス番号の決定 (整数形式), ファイルの読み取りおよび書き込みの監視

stat -c、全体デバイス番号の決定, I/O 監視 (デバイスごと)

thread_indent(), サンプル使用例, 呼び出し先の追跡

timer.ms(), サンプル使用例, 関数呼び出しのカウンタ

timer.s(), サンプル使用例, ポーリングアプリケーションの監視, 最もよく使われるシステムコールの追跡

trigger function, 呼び出し先の追跡

usrdev2kerndev(), I/O 監視 (デバイスごと)

カーネルおよびユーザースペースで費やした時間の判定, カーネルおよびユーザースペースで費やした時間の判定

システムコールの監視, 最もよく使われるシステムコールの追跡

システムコールの監視 (プロセスごとのボリューム), プロセスごとのシステムコールボリュームの追跡

ディスク I/O トラフィックの要約, ディスク読み取り/書き込みトラフィックの要約

デバイス I/O の監視, I/O 監視 (デバイスごと)

ネットワークのプロファイリング, ネットワークのプロファイリング, カーネルでのネットワークパケットドロップの監視

ネットワークソケットコードで呼び出された関数の追跡, ネットワークソケットコードで呼び出された関数の追跡

ファイルの読み取りおよび書き込みの監視, [ファイルの読み取りおよび書き込みの監視](#)
ファイルデバイス番号 (整数形式), [ファイルの読み取りおよび書き込みの監視](#)
ファイル属性の変更の監視, [ファイル属性の変更の監視](#)
プロセスのデッドロック (futex 競合により発生), [競合ユーザースペースのロックの特定](#)
ポーリングアプリケーションの監視, [ポーリングアプリケーションの監視](#)
全体デバイス番号 (コマンドライン引数として使用), [I/O 監視 \(デバイスごと\)](#)
呼び出し先の追跡, [呼び出し先の追跡](#)
着信 TCP 接続の監視, [着信 TCP 接続の監視](#)
競合ユーザースペースのロック (futex 競合) の特定, [競合ユーザースペースのロックの特定](#)
累積 I/O の追跡, [累積 I/O の追跡](#)
複数のコマンドライン引数、例, [呼び出し先の追跡](#)
関数呼び出しの集計, [関数呼び出しのカウント](#)

SystemTap スクリプトの実行

用途, [SystemTap スクリプトの実行](#)

SystemTap スクリプト関数, [Systemtap ハンドラー/ボディー](#)

SystemTap ステートメント

SystemTap ハンドラーコンストラクト

global, [変数](#)

コマンドラインの引数, [コマンドラインの引数](#)

変数, [変数](#)

条件付き (conditional) ステートメント, [条件付き \(conditional\) ステートメント](#)

For ループ, [条件付き \(conditional\) ステートメント](#)

if/else, [条件付き \(conditional\) ステートメント](#)

While ループ, [条件付き \(conditional\) ステートメント](#)

条件演算子, [条件付き \(conditional\) ステートメント](#)

SystemTap セッション, [アーキテクチャー](#)

SystemTap ハンドラー

SystemTap ハンドラーコンストラクト, [基本的な SystemTap ハンドラーコンストラクト](#)
構文および形式, [基本的な SystemTap ハンドラーコンストラクト](#)

systemtap パッケージ, [SystemTap のインストール](#)

systemtap-runtime パッケージ, [SystemTap のインストール](#)

systemtap-testsuite パッケージ

サンプルスクリプト, [便利な SystemTap スクリプト](#)

T

Tapsets

定義, [Tapsets](#)

target()

関数, [Systemtap ハンドラー/ボディー](#)

TCP 接続 (着信)、監視

SystemTap スクリプトの例, [着信 TCP 接続の監視](#)

thread_indent()

関数, [Systemtap ハンドラー/ボディー](#)

thread_indent(), サンプル使用例

SystemTap スクリプトの例, [呼び出し先の追跡](#)

tid()

関数, [Systemtap ハンドラー/ボディー](#)

timer イベント

イベント, [イベント](#)

timer.ms(), サンプル使用例

SystemTap スクリプトの例, [関数呼び出しのカウンタ](#)

timer.s(), サンプル使用例

SystemTap スクリプトの例, [ポーリングアプリケーションの監視, 最もよく使われるシステムコールの追跡](#)

tracepoint, [イベント](#), [カーネルでのネットワークパケットドロップの監視](#)

trigger function

SystemTap スクリプトの例, [呼び出し先の追跡](#)

U

uid()

関数, [Systemtap ハンドラー/ボディー](#)

uname -m, [必要なカーネル情報パッケージのインストール](#)

uname -r, [必要なカーネル情報パッケージのインストール](#)

usrdev2kerndev()

SystemTap スクリプトの例, [I/O 監視 \(デバイスごと\)](#)

V

VF 読み取り、集計

関連する値の増加

アレイ演算, [関連する値の増加](#)

vfs.file_operation

イベント, [イベント](#)

W

While ループ

条件付き (conditional) ステートメント

ハンドラー, [条件付き \(conditional\) ステートメント](#)