Red Hat Enterprise Linux 7

SystemTap ビギナーズガイド

SystemTap 入門

Last Updated: 2021-01-15

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

SystemTap 入門

William Cohen
Red Hat Software Engineering
wcohen@redhat.com

Don Domingo
Red Hat Customer Content Services

Vladimír Slávik
Red Hat Customer Content Services
vslavik@redhat.com

Robert Kratky
Red Hat Customer Content Services

Jacquelynn East
Red Hat Customer Content Services

法律上の通知

概要

SystemTap ビギナーズガイド では、SystemTap を使用して Red Hat Enterprise Linux 7 の各種サブシステムを監視する基本的な方法を詳細に説明します。本書は、RHCSA 試験を受けたことのあるユーザー、もしくは Red Hat Enterprise Linux 7 で同様のレベルの専門知識をお持ちのユーザーを対象としています。

目次

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ービングのツールです。**netstat**、**ps**、**top**、**iostat** などのツールの出力に似た情報を提供します。ただし、SystemTap は、収集した情報に対して、より多くのフィルタリングと分析オプションを提供するように設計されています。

システム管理者は、SystemTap を、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。特に、他の同様のツールがシステムのボトルネックを特定できず、システムアクティビティーを深く分析する必要がある場合に有用です。同様に、アプリケーション開発者も、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と、インストルメント化、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あまり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。

これに対し、『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 の概要を紹介し、ユーザーがアーキテクチャーに慣れるように、全カーネルタイプの設定手順を提供します。
- 異なるコンポーネントのシステム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を提供します。

1.2. SYSTEMTAP の機能

SystemTap は当初、**dprobes** や Linux Trace Toolkit のような以前の Linux プロービング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、ユーザーにカーネルのアクティビティーを追跡するインフラストラクチャーを提供することで、既存の Linux 監視ツールを補完することを目的としています。さらに、SystemTap は、この機能を以下の 2 つの特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。このスクリプトは、SystemTap のアプリケーションの例を提供するために使われており、ユーザーは、独自の SystemTap スクリプト作成時に、この機能を参考にできます。

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 のデプロイにインストールが必要となるのは、SystemTap パッケージと対応するカーネルの -devel、-debuginfo および -debuginfo-common-*arch* パッケージです。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**各** 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 パッケージと -debug パッケージを混同しがちです。SystemTap のデプロイに必要となるのは、カーネルの -debuginfo パッケージのインストールであって、カーネルの -debug バージョンではない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** で以下のコマンドを実行して systemtap と systemtap-runtime の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。この情報により SystemTap はインストルメンテーションのコード生成が可能になります。この情報は、一致する kernel-devel、kernel-debuginfo、および kernel-debuginfo-common-*arch* パッケージに含まれています (ここでの *arch* は、ご使用のシステムのハードウェアプラットフォームになります。これは、**uname -m** コマンドを実行すると判明します)。

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

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

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

上記のコマンドでは、ご使用中の Red Hat Enterprise Linux システムのバリアントによって、*variant* を **server**、**workstation**、または **client** に置き換えます。バリアントを確認するには、以下のコマンドを実行します。

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

SystemTap がプローブするカーネルは、kernel-devel、kernel-debuginfo、および kernel-debuginfo-common-*arch* パッケージのバージョン、バリアント、およびアーキテクチャーと**正確に**一致する必要があります。システムで実行中のカーネルを確認するには、以下のコマンドを実行します。

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

たとえば、AMD64 または Intel 64 マシン上のカーネルバージョン **3.10.0-327.4.4.el7** に SystemTap を使用するには、以下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、SystemTap を実行するために必要なパッケージを現行カーネルにインストールするには、**root** で以下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、以下のコマンドを実行します。

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

このコマンドは単に、SystemTap に **read performed** をプリントして、仮想ファイルシステムの読み込みが検出されると、正常に終了するよう指示します。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ルメンテーションを正常に作成できたこと、インスト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 (テキストをプリントし、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ー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 台のコンピューター上の SystemTap スクリプトから別のコンピューターで使用する SystemTap インスト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 インストルメンテーションモジュールを使用するために各**ターゲットマシン**にインストールする必要があるのは 1 つのパッケージ systemtap-runtime のみです。

重要

構築された**インスト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。

インストルメンテーションモジュール

SystemTap スクリプトから構築されるカーネルモジュールです。**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

このシステム上で (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がコンパイルされ、**ターゲットシステム** に読み込まれます。

ターゲットシステム

このシステム内で (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**が構築されます。

ターゲットカーネル

ターゲットシステムのカーネルです。このカーネルが**インスト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. **systemtap-runtime** パッケージを各**ターゲットシステム**に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、**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」 にある方法で**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
```

ここでの *kernel_version* は、**ターゲットカーネル**のバージョンを指します (ターゲットマシンで実行した **uname -r** の出力)。*script* は、**インストルメンテーションモジュール**に変換されるスクリプトです。*module_name* は、希望する **インストルメンテーションモジュール**の名前です。

インスト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、**simple.stp** という名前の SystemTap スクリプトから **3.10.0-327.4.4.el7 ターゲットカーネル** に **simple.ko** という **インストルメンテーションモジュール**を作成するには、以下のコマンドを実行します。

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

これで **simple.ko** という名前のモジュールが作成されます。**simple.ko インストルメンテーションモジュール** 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 **stap** コマンドで実行されます。**stap** を使うと、SystemTap スクリプトを標準入力またはファイルから実行できます。

stap と **staprun** を実行するには、システム対する権限の昇格が必要になります。ただし、すべてのユーザーが SystemTap 実行のために **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、その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。

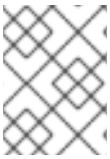
stapdev

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 インストルメンテーションモジュールを実行することができます。

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、この権限は実質的には **root** アクセスを **stapdev** メンバーに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は **staprun** を使用して SystemTap インストルメンテーションモジュールの実行ができるだけです。また、このモジュールの実行が可能なのは、**/lib/modules/*kernel_version*/systemtap/** からのみです。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** ユーザーのみとする必要があることに注意してください。



注記

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。

stap で使用する一般的なオプションには、以下のものがあります。

-v

SystemTap セッションの出力を詳細なものにします。このオプション (たとえば、**stap -vvv script.stp**) は反復してスクリプトの実行に関するより多くの詳細を提供することができます。スクリプトの実行時にエラーが発生すると、これはより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情報は、*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()** に関する詳細情報は、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。指定されたコマンドへのパスは、**cp** ではなく **/bin/cp** (**stap *script* -c /bin/cp** にあるように) のように、完全パスを使用する必要があります。**target()**に関する詳細情報は、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

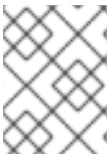
```
echo "probe timer.s(1) {exit()}" | stap -
```

例2.1「標準入力からスクリプトを実行」 では、**echo** が標準入力に渡したスクリプトを **stap** に実行するように指示しています。**stap** で使用するオプションはすべて、**-** スイッチの前に挿入します。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 の詳細は、**man stap** を参照してください。

SystemTap インストルメンテーション (クロスインストルメンテーション中に 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** を使用します。**staprun** およびクロスインストルメンテーションの詳細は、「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」 を参照してください。



注記

stap オプションの **-v** と **-o** は、**staprun** でも機能します。**staprun** の詳細は、man ページの staprun(1) 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という 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ーを使用してスクリプトの出力を保存します。次に、SystemTap インストルメンテーションモジュールが読み込まれてプローブが開始し、インストルメンテ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、インストルメンテーションは再度アタッチされ、メモリーバッファー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ー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
```

カーネルバッファーはデフォルトでは 1MB のサイズですが、**-s** オプションを使用してバッファー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) 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、バッファーを 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。ファイルのサイズと数は、**-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続けて指定します。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
```

コマンドがプリントする数字は、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。このため、**ls -sh /tmp/pfaults.log.*** を実行して表示されるのは、2 ファイルのみです。

```
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、再使用して実行中の Linux システムのアクティビティーを深く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。イベントが発生したら、Linux カーネルがハンドラーをサブルーチンとして実行し、その後に通常の動作を再開します。

イベントには関数の開始や終了、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、結果のプリントなどが含まれます。

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 が既存の tapset ライブラリー (通常は **/usr/share/systemtap/tapset/** ディレクトリー内) の使用された tapsets に対してスクリプトをチェックします。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 (詳細は 「SystemTap のインストール」 を参照してください)。

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 パッケージ内の **staprun** がこの機能を提供します (詳細は 「SystemTap のインストール」 を参照してください)。

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。

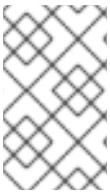
5. SystemTap セッションが終了すると、プローブは無効になり、カーネルモジュールは読み込み解除されます。

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** の詳細情報は、man ページの stap(1) を参照してください (SystemTap と正常にマシンにインストールした後)。

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構成されています。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、合わせて **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** 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プリント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。診断プリント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、特定の方法でプリントするよう指定できます。

書式

SystemTap スクリプトは **.stp** ファイル拡張子を使用し、以下の書式のプローブが含まれます。

```
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 内の *statements* は、*event* のプローブの実行時に実行されます。*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するようになります。たとえば、**close** システムコールのエントリーと終了を指定するには、それぞれ **syscall.close** と **syscall.close.return** を使用します。

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*")

function カーネル関数へのエントリー。たとえば **kernel.function("sys_open")** は、**sys_open** カーネル関数がシステム内のスレッドに呼び出される際に発生する **イベント** を指します。**sys_open** カーネル関数の 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と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 およびそれ以降) 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ー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。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** 接尾辞の使用は、**kernel.function()** の場合と同様です。例3.2「moduleprobe.stp」 のプローブハンドラーには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、有用なデータは表示されません (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。ここでの *kernel_version* は、現在読み込まれているカーネルの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ポイントは、主にカウンターやタイマー、または同様のコンストラクトで構成されています。

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 セッションの開始です。つまり、SystemTap スクリプトの実行と同時です。

end

SystemTap セッションの終了。

timer イベント

ハンドラーの定期実行を指定するイベント。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」 では、プローブが 4 秒ごとに **hello world** をプリントします。以下のような timer イベントも使用できます。

- **timer.ms(*ミリ秒*)**
- **timer.us(*マイクロ秒*)**
- **timer.ns(*ナノ秒*)**
- **timer.hz(*ヘルツ*)**
- **timer.jiffies(*jiffies*)**

timer イベントを情報を収集する他のプローブと併せて使用すると、定期的な更新が表示でき、その情報の変遷が分かります。

重要

SystemTap では、多数のプローブイベントの使用をサポートしています。サポート対象のイベントに関する情報は、man ページの stapprobes(3) 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

以下のサンプルスクリプトを見ていきましょう。

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** をプリントして改行し、終了するものです。

注記

SystemTap スクリプトは、**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。

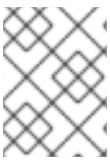
printf ( ) ステートメント

printf() ステートメントは、データをプリントする最も簡単な関数の 1 つです。 **printf()** を以下の書式で使用すると、多くの SystemTap 関数を使用するデータを表示できます。

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

format string では、*arguments* のプリント方法を指定します。例3.4「helloworld.stp」 の format string は、単に SystemTap に **hello world** のプリントを指示するだけで、書式は指定していません。

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 format string に使用することもできます。format string 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** と同一のものです。

以下のプロー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** への全エントリー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** という単語をプリントします。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()** 引数として使用可能な多くの関数をサポートしています。例3.5「variables-in-printf-statements.stp」 では、SystemTap 関数 **execname()** (カーネル関数を呼び出した、またはシステムコールを実行したプロセス名) および **pid()** (現行プロセス ID) を使用しています。

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()
: この関数はプリント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。

ここで返される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()** への最初のコールからのマイクロ秒)、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コールされたか、誰がコールしたか、各関数コールの長さが特定できます。

各コールが終わり次第、次のコールが始まれば、エントリーと終了の一致は容易ですが、ほとんどの場合では最初の関数コールの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コールが開始、終了したり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、最初のコールが終了していない場合、次の関数コールをインデントして、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プリントします。

```
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
```

このサンプル出力には、以下の情報が含まれています。

- スレッドの最初の **thread_indent()** コールからの時間 (マイクロ秒単位)。
- 関数コールを実施したプロセス名 (およびその対応 ID)。
- コールがエントリー (**<-**) か終了 (**->**) かを示す矢印。インデントがあることで、コールのエントリーと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、使用可能です。

target()

以下の 2 つのコマンドのいずれかと併せて使用します。

> **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。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*** と実行すると、(**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プリントします。

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、スクリプトの実行時に引数としてプロセス ID を渡すだけで済みます。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。これらハンドラーコンストラクトのほとんどの構文は、 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()** に設定すると (**var = gettimeofday_s()** として)、すると **var** は数字として識別され、**printf()** では整数書式の指定子 (**%d**) でプリントされます。

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する必要があります。以下で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** と **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 ++** および **count_ms ++**) は、変数の値を 1 増やすために使用されます。以下のプローブでは、100 jiffies ごとに **count_jiffies** が 1 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 script 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
```

statement1 は、***condition*** 式がゼロ以外の場合に実行されます。***statement2*** は、***condition*** 式がゼロの場合に実行されます。**else** 節 (**else** *statement2*) はオプションです。***statement1*** と ***statement2*** の両方とも、ステートメントブロックとすることができます。

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読み込み (**vfs_read**) と書き込み (**vfs_write**) 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")** の) **vfs_read** と一致する場合、スクリプトは変数 **countread** 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、これが値を変更することで ***condition*** が最終的にゼロになる必要があります。

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が異なります。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、連想アレイは一意の鍵の集合です。アレイ内のそれぞれの鍵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]
```

ここでの ***array_name*** は、アレイが使用する任意の名前になります。 ***index_expression*** は、アレイ内の特定の一意の鍵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 (一意の鍵) 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。3 人に割り当てる年齢 (関連する値) をそれぞれ 23、24、および 25 とするには、以下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、鍵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** の値を鍵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鍵に設定する非常に基本的な例を示しています。ハンドラー関数を ***index_expression*** と ***value*** の両方として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 (これを一意の鍵として使用) 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。これは一意の鍵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鍵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鍵のペアで構成されるアレイが作成されます。

この例では、**tid()** がアレイ **arr** で既に定義されている値を返すと、この演算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。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、例3.12「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」 からの) 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()** から鍵 **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 が一意の鍵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」 の場合など、数値演算) 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

アレイ内の一意の鍵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鍵 **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** アレイで鍵のペアをプリントすることです。

アレイ内の鍵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** アレイ内の一意の鍵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、一意の鍵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 秒ごとにプリントし、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** 反復をプリント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 位の反復のみをプリントします)。

3.5.5. アレイおよびアレイ要素の消去/削除

別のプローブに再使用するために、アレイ要素内の関連する値を消去したり、アレイ全体をリセットする必要がある場合もあります。「アレイ内での複数要素の処理」 の 例3.15「cumulative-vfsreads.stp」では、時間の経過とともにプロセスごとの VFS reads の増加が分かりますが、3 秒間で各プロセスが実行した VFS reads の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 数をプリントします。**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。以下の例を見てみましょう。

```
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、同様の方式でプリントします。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 秒ごとにプリントされます。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** でプリントします。

メンバーシップのテスト

特定の一意の鍵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、一意の鍵 **stapio** がアレイ **reads** に追加されたら **stapio read detected, exiting** をプリントするようにスクリプトに指示します。

3.5.7. 統計集計 (Statistical Aggregates) の計算

統計集計は、新規データを素早くかつ大量に累積する (集計ストリーム統計情報のみを保存) ことが重要な場合に、数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、各一意の鍵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。各一意の鍵 (**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。

統計集計が収集したデータを抽出するには、***@extractor(variable/array index expression)*** という構文書式を使用します。***extractor*** は以下の整数抽出のいずれかにします。

count

variable/array index expression 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** の各一意の鍵に保存されている**値の数**を返します。

sum

variable/array index expression 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** の各一意の鍵に保存されている**値すべての合計**を返します。

min

variable/array index expression に保存されているすべての値で最も小さいものを返します。

max

variable/array index expression に保存されているすべての値で最も大きいものを返します。

avg

variable/array index expression に保存されているすべての値の平均を返します。

統計集計を使用する際には、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ができます。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。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プリント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。tapsets 標準ライブラリーは、デフォルトで **/usr/share/systemtap/tapset/** ディレクトリーに格納されます。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構成します。

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。Tapsets 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、tapsets 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 パッケージのインストール後に **/usr/share/systemtap/testsuite/systemtap.examples/** ディレクトリーから利用可能となります。

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() 関数が以下の式を使用すること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** or **else** 条件です。2 番目のステートメントは、以下の擬似コードをより簡潔にしたものです。

```
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** – 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** – プロセスがデータの送受信に使用したイーサネットデバイス (例：eth0、eth1)。
- **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** – プロセ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。**probe timer.ms(5000)** を編集すると、この間隔が変更できます。例4.2「例4.1「nettop.stp」 のサンプル出力」 には、例4.1「nettop.stp」 からの 20 秒間に渡る出力を引用してあります。

例4.2 例4.1「nettop.stp」 のサンプル出力

```
[...]
  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を説明します。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、**thread_indent()** が機能する様子の説明に使用した SystemTap 関数 の 例3.6「thread_indent.stp」 と同じ ものです。

例4.4 例4.3「socket-trace.stp」 のサンプル出力

```
[...]
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」 のサンプル出力」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プリントアウトされます。

- 現行の **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」 のサンプル出力

```
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。Linux カーネルにはトレースポイント **kernel.trace("kfree_skb")** を含むものもあり、これは簡単にパケットが破棄された場所を追跡します。例4.7「dropwatch.stp」 では、**kernel.trace("kfree_skb")** を使用して、パケットの破棄を追跡します。このスクリプトは、パケットが破棄された場所を 5 秒ごとに要約します。

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ーへのポインター (**$skb**) と解放されているバッファーのカーネルコード内での場所 (**$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」 のサンプル出力」 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の数が記載されています。

例4.8 例4.7「dropwatch.stp」 のサンプル出力

```
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-$(uname -r)** ファイルを確認します。このファイルには、各関数の開始アドレスが記載されており、例4.8「例4.7「dropwatch.stp」 のサンプル出力」 の出力のアドレスを特定の関数名にマップできます。以下の **/boot/System.map-$(uname -r)** ファイルの抜粋を使用して、アドレス 0xffffffff8024cd0f を関数 **unix_stream_recvmsg** に、アドレス 0xffffffff8044b472 を関数 **arp_rcv** にマッピングします。

```
[...]
ffffffff8024c5cd T unlock_new_inode
ffffffff8024c5da t unix_stream_sendmsg
ffffffff8024c920 t unix_stream_recvmsg
ffffffff8024cea1 t udp_v4_lookup_longway
[...]
ffffffff8044addc t arp_process
ffffffff8044b360 t arp_rcv
ffffffff8044b487 t parp_redo
ffffffff8044b48c 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 アクティビティー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、ディスクに重い読み取りまたは書き込みを行なっているプロセス上位 10 位を出力します。例4.10「例4.9「disktop.stp」 のサンプル出力」 はこのスクリプトのサンプル出力となり、記載されているプロセスについて以下のデータが含まれます。

- **UID** – ユーザー ID。ユーザー ID が **0** の場合は、root ユーザーを指します。
- **PID** – プロセスの ID。
- **PPID** – プロセスの **親プロセス**の ID。
- **CMD** – プロセスの名前。
- **DEVICE** – プロセスが読み取りまたは書き込みを行な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** – ディスクから読み取った、またはディスクに書き込んだデータ量。

例4.9「disktop.stp」 の日時に関する出力は、**ctime()** と **gettimeofday_s()** の関数で返されます。**ctime()** は、Unix 時間 (1970 年 1 月 1 日) 以降の秒単位経過をカレンダー時刻で引き出します。**gettimeofday_s()** は、Unix 時間以降の**実際の**秒数をカウントします。これは、出力についてかなり正確でヒューマンリーダブルなタイムスタンプを提供します。

このスクリプトでは、**$return** は各プロセスが仮想ファイルシステムから読み取ったまたは書き込んだ実際のバイト数を保存するローカル変数です。**$return** は return プローブ(**vfs.read.return** や **vfs.read.return** など) での使用のみが可能です。

例4.10 例4.9「disktop.stp」 のサンプル出力

```
[...]
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 **$count** を使用してシステムコールが読み取りまたは書き込みを**試みた** データ量 (バイト単位) も追跡します。(「ディスク読み取り/書き込みトラフィックの要約」 の例4.9「disktop.stp」 で使用されている) **$return** は、**実際に** 読み取り/書き込みされたデータ量を保存することに注意してください。**$count** を使用できるのは、(**syscall.read** や **syscall.write** など) データの読み取りや書き込みを追跡するプローブのみです。

例4.12 例4.11「iotime.stp」 のサンプル出力

```
[...]
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」 のサンプル出力」 では、以下のデータがプリントアウトされます。

- 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** の行のペアが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 stap
# 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プリントします。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。この情報は降順で 1 秒ごとに追跡、プリントアウトされます。

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** が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」 のサンプル出力

```
[...]
          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 アクティビティー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。ここでの *directory* は、監視するデバイス上にあるものにします。

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()** による出力は、**MKDEV()**、**MINOR()**、および **MAJOR()** の関数と併せて、特定デバイスのメジャーおよびマイナー番号の決定に使用されます。

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。

以下の例は、**stap traceio2.stp 0x805** の出力の抜粋です。ここでの **0x805** は、**/home** の全体デバイス番号です。**/home** は **/dev/sda5** に存在しており、これが監視対象のデバイスになります。

例4.16 例4.15「traceio2.stp」 のサンプル出力

```
[...]
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。

これらの情報を取得するには、**stat -c '%D %i'** ***filename*** を使用します。ここでの *filename* は、絶対パスになります。

たとえば、**/etc/crontab** ファイルを監視するには、まず **stat -c '%D %i' /etc/crontab** を実行します。これで以下の出力が得られます。

```
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** 番号になります。**/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、ベース 16 の値を示します)。

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、実行している関数 (**vfs_read** または **vfs_write**)、デバイス番号 (16 進法形式)、および **inode** 番号が含まれます。例4.18「例4.17「inodewatch.stp」 のサンプル出力」 は、**stap inodewatch.stp 0x8 0x05 1078319** の出力になります (スクリプトの実行中に **cat /etc/crontab** を実行した場合)。

例4.18 例4.17「inodewatch.stp」 のサンプル出力

```
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」 の場合と似ていますが、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の ID (**uid()**) が含まれている点が異なります。例4.20「例4.19「inodewatch2-simple.stp」 のサンプル出力」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」 のサンプル出力

```
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、関数コールを監視することでカーネルアクティビティー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. 関数コールのカウント

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」 のサンプル出力」は、**stap functioncallcount.stp "*@mm/*.c"** の出力抜粋になります。

例4.22 例4.21「functioncallcount.stp」 のサンプル出力

```
[...]
__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コール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。

- その開始と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()** に関する詳細情報は、SystemTap 関数 のエントリーを参照してください。

以下は、**stap para-callgraph.stp 'kernel.function("*@fs/*.c")' 'kernel.function("sys_read")'** の出力抜粋例になります。

例4.24 例4.23「para-callgraph.stp」 のサンプル出力

```
[...]
   267 gnome-terminal(2921): <-do_sync_read return=0xfffffffffffffff5
   269 gnome-terminal(2921):<-vfs_read return=0xfffffffffffffff5
     0 gnome-terminal(2921):->fput file=0xffff880111eebbc0
     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=0xfffffffffffffff5
    18 gnome-terminal(2921):<-vfs_read return=0xfffffffffffffff5
     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」 のサンプル出力」 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」 のサンプル出力

```
  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 の使用量および省電力を改善す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容疑者を特定でき、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」 のサンプル出力

```
  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| multiload-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」 のサンプル出力」 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 秒あたり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」 のサンプル出力」 がサンプル出力になります。

例4.30 例4.29「topsys.stp」 のサンプル出力

```
------------------------------------------------------------
                  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 1 番多く特定の「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と他の古いリソース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」 のサンプル出力」 はサンプル出力になります。

例4.32 例4.29「topsys.stp」 のサンプル出力

```
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もできます。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し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と、**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表示されます。

- 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」 のサンプル出力」 は 例4.34「futexes.stp」 (約 20 秒) を終了した際の出力の抜粋です。

例4.35 例4.34「futexes.stp」 のサンプル出力

```
[...]
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

SystemTap を使用する上で最もよく遭遇する可能性のあるエラーを説明しています。

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、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。SystemTap は間違ったコンストラクトのタイプを検出しプローブのコンテキストを表示します。

次の無効な 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 では目的にあった tapsets がない場合にはスクリプトに C コードを埋め込むことができるため便利ですが、埋め込みの C コンストラクトは安全ではありません。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て **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、**-g** オプションを使用して **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** 関数で間違ったタイプを使用しています (**%s** または **%d**)。このエラーを 例 5.1「error-variable.stp」 に示します。**execname()** 関数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。次のような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コール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か関数へのパラメーター数を見ることができます。

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す)。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

var アレイがアクティブな **foreach** ループ内で修正されています (割り当てられるまたは削除される)。このエラーは、スクリプト内の演算が **foreach** ループ内で関数コール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** にマッチ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*

pnt イベントとハンドラー関数が各種の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 **kernel.function("*name*")** イベントが含まれ、*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ー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により発生します。インストールされた kernel-debuginfo パッケージ自体に一貫性もしく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*

エラー、プローブのスキップ、もしくはこれら両方が実行中に発生しました。*N* はエラー数で、*M* は、インターバルの間にイベントハンドラーを実行する時間が足りなかったために実行されなかったなどの理由でスキップされたプローブ数です。

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

集計値を含むアレイに明確な鍵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 の開発、使用、および導入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』 を元にしています。『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 ページの下の方には **stapprobes.scsi**、**stapprobes.kprocess**、**stapprobes.signal** など特定のシステムコンポーネント用に似たようなプローブポイントを列挙している別の man ページの一覧があります。

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** ので注意してください。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)

SystemTap の言語構成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 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。

testsuite (テストスイート)

systemtap-testsuite パッケージを使用するとソースから構築しなくても SystemTap ツールチェーン全体の検証を行うことができます。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

systemtap-testsuite に収納されているサンプルスクリプトは、デフォルトでは **/usr/share/systemtap/testsuite/systemtap.examples** に格納されています。

付録A 改訂履歴

改訂 7-7 Mon Aug 5 2019 Vladimír Slávik
Red Hat Enterprise Linux 7.7 GA 公開用のリリース。

改訂 7-6 Tue Oct 30 2018 Vladimír Slávik
Red Hat Enterprise Linux 7.6 GA 公開用のリリース。

改訂 7-5.1 Mon Jun 18 2018 Radek Bíba
stap テスト画面に id を使用します。

改訂 7-5 Tue Jan 09 2018 Vladimír Slávik
Red Hat Enterprise Linux 7.5 Beta 公開用リリース。

改訂 7-4 Wed Jul 26 2017 Vladimír Slávik
Red Hat Enterprise Linux 7.4 公開用リリース。

改訂 7-3.9 Tue May 16 2017 Robert Krátký
7.4 ベータリリース向けのビルド

改訂 1-8 Wed Oct 19 2016 Robert Kratky
Red Hat Enterprise Linux 7.3 用のリリース。

改訂 1-6 Wed Jan 20 2016 Robert Kratky
多数の修正のある非同期リリース。

改訂 1-5 Thu Nov 11 2015 Robert Kratky
Red Hat Enterprise Linux 7.2 用のリリース。

改訂 0-3 Fri Dec 6 2013 Jacquelynn East
Red Hat Enterprise Linux 7.0 用のリリース。

索引

シンボル

A

B

C

G

I

T

U

V

W