Red Hat Enterprise Linux 6

SystemTap Tapset リファレンス

Red Hat Enterprise Linux 6 SystemTap 用

Last Updated: 2020-07-13

Red Hat Enterprise Linux 6 SystemTap Tapset リファレンス

Red Hat Enterprise Linux 6 SystemTap 用

Robert Krátký
Red Hat Customer Content Services
rkratky@redhat.com

William Cohen
Red Hat Performance Tools

Don Domingo
Red Hat Customer Content Services

Red Hat, Inc.

編集者

Jacquelynn East
Red Hat Customer Content Services

法律上の通知

概要

tapset リファレンスガイドは、ユーザーが SystemTap スクリプトに適用できる最も一般的な tapset定義について説明しています。本ガイドに記載されている tapset はすべて、SystemTap の最新のアップストリームバージョンとして説明されています。

目次

前書き

1. ドキュメント規則

本ガイドでは、いくつかの規則を使用して特定の単語やフレーズを強調表示し、特定の情報への注意を促しています。

1.1. 誤字規則

特定の単語や句に注意を促すために 4 つの誤字規則を使用しています。これらの規則や、これらが適用される状況は以下のとおりです。

Mono-spaced Bold

シェルコマンド、ファイル名、パスなど、システム入力を強調表示するために使用されます。キーとキーの組み合わせを強調表示するのにも使用されます。以下に例を示します。

> 現在の作業ディレクトリーのファイル **my_next_bestselling_novel** のファイルの内容を表示するには、シェルプロンプトで **cat my_next_bestselling_novel** コマンドを入力し、**Enter** を押してコマンドを実行します。

上記には、ファイル名、シェルコマンドおよびキーが含まれます。これはすべて mono-spaced bold で示され、コンテキストにより区別可能なものになります。

キーの組み合わせは、キーの組み合わせの各パーツをつなげるプラス記号によって個別のキーと区別できます。以下に例を示します。

> **Enter** を押してコマンドを実行します。
>
> **Ctrl**+**Alt**+**F2** を押して、仮想端末に切り替えます。

最初の例では、押す特定のキーを強調表示しています。2 つ目の例は、同時に押す 3 つのキーのセットというキーの組み合わせを強調表示しています。

ソースコードについて記述では、クラス名、メソッド、関数、変数名、および段落内で記述された戻り値は、**mono-spaced bold**で示されます。以下に例を示します。

> ファイル関連クラスには、ファイルシステムの **filesystem**、ファイルの **file**、ディレクトリーの **dir** が含まれます。各クラスには、独自の関連付けられたパーミッションセットがあります。

Proportional Bold

これは、アプリケーション名、ダイアログボックステキスト、ラベルが付いたボタン、チェックボックスおよびラジオボタン、メニュータイトルおよびサブメニュータイトルなど、システムで発生した単語またはフレーズを示します。以下に例を示します。

> メインメニューバーから System → Preferences → Mouse を選択して、Mouse Preferences を起動します。**Buttons** タブで、**Left-handed mouse** のチェックボックスを選択し、**Close** をクリックして、左から右に主要なマウスボタンを切り替えます (左側のマウスは適切なマウスになります)。

特殊文字を gedit ファイルに挿入するには、メインメニューバーから **Applications → Accessories → Character Map** を選択します。次に、**Character Map** メニューバーから **Search → Find...** を選択し、**Search** フィールドに文字の名前を入力して **Next** をクリックします。目的の文字は **Character Table** に強調表示されます。この強調表示した文字をダブルクリックして、**Text to copy** フィールドに配置し、**Copy** ボタンをクリックします。次に、ドキュメントに戻り、gedit メニューバーから **Edit → Paste** を選択します。

上記のテキストにはアプリケーション名、システム全体のメニュー名および項目、アプリケーション固有のメニュー名、GUI インターフェース内のボタンおよびテキストなどがあります。すべては proportional bold で示され、コンテキストと区別できます。

Mono-spaced Bold Italic または *Proportional Bold Italic*

mono-spaced bold または proportional bold のいずれでも、イタリックを追加すると、置換または変数テキストが表示されます。イタリックは、状況に応じて変化するテキストや、文字を入力しないテキストを表します。以下に例を示します。

ssh を使用してリモートマシンに接続するには、シェルプロンプトで **ssh *username@domain.name*** を入力します。リモートマシンが **example.com** で、そのマシンのユーザー名が john の場合は、**ssh john@example.com** と入力します。

mount -o remount *file-system* コマンドは、名前付きのファイルシステムを再マウントします。たとえば、**/home** ファイルシステムを再マウントする場合、コマンドは **mount -o remount /home** になります。

現在インストールされているパッケージのバージョンを表示するには、**rpm -q *package*** コマンドを使用します。結果は以下のようになります: ***package-version-release***。

上記の太字のイタリック体の用語、username、domain.name、file-system、package、version、および release に注意してください。各単語はプレースホルダーで、コマンドの発行時に入力するテキストまたはシステムによって表示されるテキストのどちらかになります。

作業のタイトルを示す標準的な使用法のほかに、イタリックは新用語と重要な用語の最初の使用を示します。以下に例を示します。

Publican は *DocBook* 公開システムです。

1.2. 引用規則

端末の出力およびソースコードの一覧は、周りのテキストから視覚的に表示されます。

端末に送信される出力は **mono-spaced roman** にセットされて表現されます。

```
books       Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

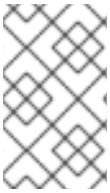
ソースコードの一覧も **mono-spaced roman** にセットされますが、構文の強調表示が以下のように追加されます。

```
static int kvm_vm_ioctl_deassign_device(struct kvm *kvm,
                 struct kvm_assigned_pci_dev *assigned_dev)
{
        int r = 0;
        struct kvm_assigned_dev_kernel *match;
```

```
        mutex_lock(&kvm->lock);

        match = kvm_find_assigned_dev(&kvm->arch.assigned_dev_head,
                                      assigned_dev->assigned_dev_id);
        if (!match) {
                printk(KERN_INFO "%s: device hasn't been assigned before, "
                  "so cannot be deassigned\n", __func__);
                r = -EINVAL;
                goto out;
        }

        kvm_deassign_device(kvm, match);

        kvm_free_assigned_device(kvm, match);

out:
        mutex_unlock(&kvm->lock);
        return r;
}
```

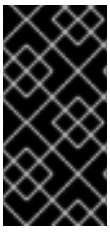
1.3. 注記および警告

最後に、3 つの視覚的スタイルを使用して、見落とす可能性のある情報に注意を促します。



注記

注記とは、タスクへのヒント、ショートカット、または代替アプローチです。注意を無視しても悪い結果を招くことはありませんが、便利なヒントを見逃してしまう可能性があります。



重要

見落としやすい詳細のある重要なボックス: 現行セッションにのみ適用される設定変更や、更新を適用する前に再起動が必要なサービスなどです。「Important」というラベルが付いたボックスを無視しても、データが失われることはありませんが、スムーズな操作が行えないことがあります。

警告

警告は無視すべきではありません。警告を無視すると、データが失われる可能性があります。

2. ヘルプの利用とフィードバック提供

2.1. ヘルプが必要ですか?

本書で説明されている手順で問題が発生した場合は、Red Hat カスタマーポータル http://access.redhat.com にアクセスしてください。カスタマーポータルでは、以下を行うことができます。

- Red Hat 製品に関する技術サポート記事の知識ベースの検索または閲覧。
- Red Hat グローバルサポートサービス (GSS) へのサポートケースの送信。
- その他の製品ドキュメントへのアクセス。

Red Hat は、Red Hat のソフトウェアおよびテクノロジーについて、多くの電子メーリングリストも提供しています。一般に公開されているメーリングリストの一覧は、https://www.redhat.com/mailman/listinfoを参照してください。メーリングリストの名前をクリックして、その一覧をサブスクライブするか、またはメーリングリストのアーカイブにアクセスします。

2.2. ご意見をお寄せください

本書で誤字・脱字を発見されたり、このマニュアルを改善するための提案をお持ちの場合は、弊社までご連絡ください。Bugzilla でレポートを送信してください。http://bugzilla.redhat.com/ の Red_Hat_Enterprise_Linux の製品。

バグレポートを送信する際には、マニュアル識別子(『doc-SystemTap_Tapset_Reference』) を指定してください。

本書を改善するためのご意見やご提案をお寄せいただく場合は、できるだけ具体的にご説明ください。エラーが見つかった場合は、簡単に確認できるように、セクション番号と前後のテキストを含めてください。

第1章 はじめに

SystemTap はフリーソフトウェア (GPL) インフラストラクチャーを提供し、実行中の Linux システムに関する情報の収集を簡素化します。これにより、パフォーマンスまたは機能的な問題の診断が容易になります。SystemTap により、開発者はデータの収集に必要になる可能性のある未知で破壊的なインストルメント、再コンパイル、インストール、および再起動を行う必要がなくなります。

SystemTap では、ライブで実行中のカーネル用にインストルメンテーションを記述する簡単なコマンドラインインターフェースとスクリプト言語を利用できます。このインストルメンテーションは、*tapset* ライブラリーで提供されるプローブポイントと関数を使用します。

簡単に説明すると、tapset は、カーネルサブシステムに関する知識を他のスクリプトで使用できる事前記述されたプローブおよび関数にカプセル化するスクリプトです。tapsets は C プログラムのライブラリーに類似しています。これらのライブラリーは、カーネルエリアの基礎となる詳細を非表示にし、カーネルの管理および監視に必要な主な情報を公開します。これらは通常、カーネルの専門家によって開発されます。

tapset は高レベルなデータとサブシステムの状態遷移を公開します。多くの場合、優れた tapset の開発者であれば、SystemTap ユーザーがカーネルサブシステムの低レベルの詳細をほとんど認識していないと想定しています。そのため、tapset の開発者は、通常の SystemTap ユーザーが有用で便利な SystemTap スクリプトを記述するのに役立つ tapset を記述します。

1.1. 本ガイドの目的

本ガイドは、SystemTap の最も有用で一般的な tapset エントリーを取り上げることを目的としています。また、適切な tapset 開発やドキュメントに関するガイドラインも記載しています。本ガイドに含まれる tapset の定義は、各 tapset ファイルのコードの適切にフォーマットされたコメントから自動的に抽出されます。そのため、本ガイドの定義に対する改訂は、それぞれの tapset ファイルに直接適用する必要があります。

第2章 TAPSET 開発ガイドライン

本章では、適切な tapset ドキュメントのアップストリームのガイドラインについて説明します。また、本ガイドで適切に定義されるように tapset を適切に記述する方法も含まれています。

2.1. TAPSET の記述

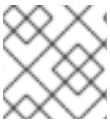
適切な tapset を記述するための最初の手順は、サブジェクトエリアの単純なモデルを作成することです。たとえば、プロセスサブシステムのモデルには以下が含まれる場合があります。

キーデータ

- プロセス ID
- 親プロセス ID
- プロセスグループ ID

状態遷移

- forked (フォーク)
- exec'd (実行)
- running (実行中)
- stopped (停止)
- terminated (終了)



注記

上記のリストは両方とも例であり、完全なリストではありません。

サブシステムの知識を使用して、モデルの要素を公開するプローブポイント (関数エントリーおよび終了) を検索し、それらのポイントのプローブエイリアスを定義します。一部の状態遷移は複数の場所で発生する可能性があることに注意してください。この場合、エイリアスはプローブを複数の場所に配置できます。

たとえば、プロセス実行は、関数 **do_execve()** または **compat_do_execve()** のいずれかで発生する可能性があります。以下のエイリアスは、これらの関数の最初にプローブを挿入します。

```
probe kprocess.exec = kernel.function("do_execve"),
kernel.function("compat_do_execve")
{probe body}
```

プローブを可能な限り安定したインターフェース (インターフェースレベルで変更できない関数など) に配置してみてください。これにより、カーネルの変更により tapset が破損する可能性が低くなります。カーネルのバージョンまたはアーキテクチャーの依存関係を回避できない場合は、プリプロセッサー条件を使用します (詳細は、man ページの **stap(1)** を参照してください)。

プローブポイントで利用可能なキーデータでプローブボディーを入力します。関数エントリープローブは、関数に指定されたエントリーパラメーターにアクセスできますが、終了プローブはエントリーパラメーターおよび戻り値にアクセスできます。適切な場合はデータを意味のある形式に変換します (バイ

トをキロバイト、状態値は文字列など)。

補助関数を使用して一部のデータにアクセスしたり、変換する必要がある場合があります。補助関数は多くの場合、埋め込み C を使用して SystemTap 言語では実行できないことを行います。たとえば、一部のコンテキスト内の構造フィールドへのアクセス、リンクされた一覧などです。他の tapset で定義された補助関数を使用するか、独自の tapset を作成することができます。

以下の例では、新しいプロセスに対して **task_struct** のポインターを **copy_process()** に返します。新しいプロセスのプロセス ID は **task_pid()** ポインターを呼び出して、**task_struct** を渡すことで取得されます。この場合、補助関数は **task.stp** で定義されている埋め込み C 関数です。

```
probe kprocess.create = kernel.function("copy_process").return
{
  task = $return
  new_pid = task_pid(task)
}
```

すべての関数にプローブを作成することは推奨されません。多くの SystemTap ユーザーには必要ありませんし、理解する必要もありません。簡単で高レベルな tapset を維持します。

2.2. TAPSET の要素

以下のセクションでは、tapset を作成する最も重要な側面を説明します。ここでのコンテンツの大半は、SystemTap の tapset のアップストリームライブラリーへの貢献を希望する開発者に適しています。

2.2.1. tapset ファイル

tapset ファイルは SystemTap GIT ディレクトリー ***src*/tapset/** に保存されます。ほとんどの tapset ファイルはこのレベルで保持されます。特定のアーキテクチャーまたはカーネルバージョンでのみ機能するコードがある場合は、tapset を適切なサブディレクトリーに配置することを選択できます。

インストールされている tapset は **/usr/share/systemtap/tapset/** または **/usr/local/share/systemtap/tapset** にあります。

個人用の tapset はどこにでも保存できます。ただし、SystemTap がこれらの tapset を使用できるようにするには、**-I *tapset_directory*** を使用して **stap** を呼び出す際に場所を指定する必要があります。

2.2.2. 名前空間

プローブエイリアス名は、***tapset_name.probe_name*** の形式で指定する必要があります。たとえば、シグナルを送信するプローブは、**signal.send** とすることができます。

グローバルシンボル名 (プローブ、関数、および変数) は tapset 全体で一意である必要があります。これは、複数の tapset を使用するスクリプトで名前空間の競合を回避するのに役立ちます。これを確認するには、グローバルシンボルで tapset 固有のプレフィックスを使用します。

内部シンボル名にはアンダースコア (_) をプレフィックスとして使用する必要があります。

2.2.3. コメントおよびドキュメント

すべてのプローブおよび関数には、目的、提供するデータ、および実行するコンテキストを記述するコメントブロック (割り込み、プロセスなど) が含まれている必要があります。コードの読み取りでは、不明瞭なエリアでコメントを使用します。

特殊形式のコメントは、多くの tapset から自動的に抽出され、本ガイドに含まれています。これにより、tapset の貢献者が tapset を作成し、同じ場所で文書化するのに役立ちます。tapset の文書化に指定される書式は、以下のとおりです。

```
/**
 * probe tapset.name - Short summary of what the tapset does.
 * @argument: Explanation of argument.
 * @argument2: Explanation of argument2. Probes can have multiple arguments.
 *
 * Context:
 * A brief explanation of the tapset context.
 * Note that the context should only be 1 paragraph short.
 *
 * Text that will appear under "Description."
 *
 * A new paragraph that will also appear under the heading "Description".
 *
 * Header:
 * A paragraph that will appear under the heading "Header".
 **/
```

以下に例を示します。

```
/**
 * probe vm.write_shared_copy- Page copy for shared page write.
 * @address: The address of the shared write.
 * @zero: Boolean indicating whether it is a zero page
 *         (can do a clear instead of a copy).
 *
 * Context:
 *  The process attempting the write.
 *
 *  Fires when a write to a shared page requires a page copy.  This is
 *  always preceded by a vm.shared_write.
 **/
```

自動生成される **Synopsis** コンテンツをオーバーライドするには、以下を使用します。

```
 * Synopsis:
 * New Synopsis string
 *
```

以下に例を示します。

```
/**
 * probe signal.handle - Fires when the signal handler is invoked
 * @sig: The signal number that invoked the signal handler
 *
 * Synopsis:
 * <programlisting>static int handle_signal(unsigned long sig, siginfo_t *info, struct k_sigaction *ka,
 * sigset_t *oldset, struct pt_regs * regs)</programlisting>
 */
```

エントリーの **Synopsis** の内容を上書きしても必要なタグが自動的に作成されないため、この例では **<programlisting>** タグの使用を推奨しています。

コメントの DocBook XML 出力を改善するため、コメントに以下の XML タグを使用することもできます。

- **command**
- **emphasis**
- **programlisting**
- **remark** (タグ付けされた文字列が、ドキュメントの Publican ベータビルドに表示されます)

第3章 コンテキスト関数

コンテキスト関数では、イベントが発生した場所に関する追加情報を利用できます。これらの関数は、イベントが発生した場所へのバックトレースやプロセッサーの現在のレジスター値へのバックトレースなどの情報を提供します。

名前
function::print_regs – レジスターダンプを出力します。

概要

```
function print_regs()
```

引数
なし

一般的な構文
print_regs

説明
この関数はレジスターダンプを出力します。

名前
function::execname – ターゲットプロセス (またはプロセスのグループ) の実行名を返します。

概要

```
function execname:string()
```

引数
なし

一般的な構文
execname:**string**

説明
ターゲットプロセス (またはプロセスのグループ) の実行名を返します。

名前
function::pid – ターゲットプロセスの ID を返します。

概要

```
function pid:long()
```

引数
なし

一般的な構文
pid:**long**

説明
この関数は、ターゲットプロセスの ID を返します。

名前
function::tid – ターゲットプロセスのスレッド ID を返します。

概要

```
function tid:long()
```

引数
なし

一般的な構文
tid:**long**

説明
この関数はターゲットプロセスのスレッド ID を返します。

名前
function::ppid – ターゲットプロセスの親プロセスのプロセス ID を返します。

概要

```
function ppid:long()
```

引数
なし

一般的な構文
ppid:**long**

説明
この関数は、ターゲットプロセスの親プロセスのプロセス ID を返します。

名前
function::pgrp – 現在のプロセスのプロセスグループ ID を返します。

概要

```
function pgrp:long()
```

引数
なし

一般的な構文
pgrp:**long**

説明
この関数は、現在のプロセスのプロセスグループ ID を返します。

名前
function::sid – 現在のプロセスのセッション ID を返します。

概要

```
function sid:long()
```

引数
なし

一般的な構文
sid:**long**

説明
プロセスのセッション ID は、セッションリーダーのプロセスグループ ID です。Kernel 2.6.0 以降では、セッション ID は signal_struct に格納されます。

名前
function::pexecname – ターゲットプロセスの親プロセスの実行名を返します。

概要

```
function pexecname:string()
```

引数
なし

一般的な構文
pexecname:**string**

説明
この関数は、ターゲットプロセスの親プロセスの実行名を返します。

名前
function::gid – ターゲットプロセスのグループ ID を返します。

概要

```
function gid:long()
```

引数
なし

一般的な構文
gid:**long**

説明
この関数は、ターゲットプロセスのグループ ID を返します。

名前
function::egid – ターゲットプロセスの実効 GID を返します。

概要

```
function egid:long()
```

引数
なし

一般的な構文
egid:**long**

説明
この関数は、ターゲットプロセスの実効 GID を返します。

名前
function::uid – ターゲットプロセスのユーザー ID を返します。

概要

```
function uid:long()
```

引数
なし

一般的な構文
uid:**long**

説明

この関数は、ターゲットプロセスのユーザー ID を返します。

名前
function::euid – ターゲットプロセスの実効 UID を返します。

概要

```
function euid:long()
```

引数
なし

一般的な構文
euid:**long**

説明
ターゲットプロセスの実効ユーザー ID を返します。

名前
function::is_myproc – ユーザー独自のプロセスで現在のプローブポイントが発生したかどうかを判断します。

概要

```
function is_myproc:long()
```

引数
なし

一般的な構文
is_myproc:**long**

説明
ユーザー独自のプロセスで現在のプローブポイントが発生した場合、この関数は 1 を返します。

名前
function::cpu – 現在の CPU 番号を返します。

概要

```
function cpu:long()
```

引数
なし

一般的な構文
cpu:**long**

説明
この関数は、現在の CPU 番号を返します。

名前
function::pp – アクティブなプローブポイントを返します。

概要

```
function pp:string()
```

引数
なし

一般的な構文
pp:**string**

説明
この関数は、現在実行しているプローブハンドラーに関連する完全に解決されたプローブポイントを返します (エイリアスやワイルドカードによる拡張を含む)。コンテキスト: 現在のプローブポイント。

名前
function::registers_valid – 現在のコンテキストの **register** および **u_register** の有効性を決定します。

概要

```
function registers_valid:long()
```

引数
なし

一般的な構文
registers_valid:**long**

説明
この関数は、**register** と **u_register**が現在のコンテキストで使用できる場合に 1 を返します。そうでない場合は 0 を返します。たとえば、**registers_valid** は begin または end プローブから呼び出されると 0 を返します。

名前
function::user_mode – プローブポイントがユーザーモードで発生するかどうかを判断します。

概要

```
function user_mode:long()
```

引数
なし

一般的な構文
user_mode:**long**

プローブポイントがユーザーモードで発生した場合は 1 を返します。

名前
function::is_return – 現在のプローブコンテキストが return プローブであるかどうかを指定します。

概要

```
function is_return:long()
```

引数
なし

一般的な構文
is_return:**long**

説明
現在のプローブコンテキストが return プローブである場合は 1 を返し、それ以外の場合は 0 を返します。

名前
function::target – ターゲットプロセスのプロセス ID を返します。

概要

```
function target:long()
```

引数
なし

一般的な構文
target:**long**

説明
この関数は、ターゲットプロセスのプロセス ID を返します。これは、stap に対する -x PID または -c CMD コマンドラインオプションと併用する場合に便利です。その使用例として、特定のプロセスでフィルターするスクリプトを作成できます。

名前
function::module_name – 現在のスクリプトのモジュール名。

概要

```
function module_name:string()
```

引数
なし

一般的な構文
module_name:**string**

説明
この関数は、stap モジュールの名前を返します。ランダムに生成される (stap_[0-9a-f]+_[0-9a-f]+) または stap -m <module_name> で設定されます。

名前
function::stp_pid – stapio プロセスのプロセス ID。

概要

```
function stp_pid:long()
```

引数
なし

一般的な構文
stp_pid:**long**

説明
この関数は、このスクリプトを起動した stapio プロセスのプロセス ID を返します。システム上では、その他の SystemTap スクリプトや stapio プロセスが実行されている可能性があります。

名前
function::stack_size – カーネルスタックのサイズを返します。

概要

```
function stack_size:long()
```

引数
なし

一般的な構文
stack_size:**long**

説明
この関数は、カーネルスタックのサイズを返します。

名前
function::stack_used – 使用されたカーネルスタックの容量を返します。

概要

```
function stack_used:long()
```

引数
なし

一般的な構文
stack_used:**long**

説明
この関数は、カーネルスタックで現在使用されている容量 (バイト数) を判断します。

名前
function::stack_unused – 現在使用可能なカーネルスタックの容量を返します。

概要

```
function stack_unused:long()
```

引数
なし

一般的な構文
stack_unused:**long**

説明
この関数は、カーネルスタックの現在の空き容量 (バイト数) を判断します。

名前
function::uaddr – 現在実行しているタスクのユーザー空間アドレス実験的

概要

```
function uaddr:long()
```

引数
なし

一般的な構文
uaddr:**long**

説明
プローブの発生時に現在のタスクが存在していたユーザー空間のアドレスを返します。現在実行中のタスクがユーザー空間スレッドではない場合やアドレスが見つからない場合は、ゼロが返されます。現在のタスクが **usymname** または **symdata** と組み合わされる場所を確認するために使用できます。多くのタスクは、カーネルを入力した VDSO にあります。FIXME - need VDSO tracking support #10080.

名前
function::cmdline_args – 現在のプロセスからコマンドライン引数を取得します。

概要

```
function cmdline_args:string(n:long,m:long,delim:string)
```

引数

n

最初に取得する引数 (ゼロはコマンド自体)。

m

最後に取得する引数 (-1 は n の後の引数すべて)。

delim

複数の引数を区切るために使用する文字列。

一般的な構文
cmdline_args:string(n:long, m:long, delim:string)

説明
現在のプロセスから、n から m までの引数を返します。n 未満の引数がある場合や、現在のプロセスから引数を取得できない場合は、空の文字列が返されます。m が n よりも小さい場合は、引数 n から始まるすべての引数が返されます。引数 0 は、通常コマンド自体です。

名前
function::cmdline_arg – コマンドライン引数を取得します。

概要

```
function cmdline_arg:string(n:long)
```

引数

n

取得する引数 (0 (ゼロ) はコマンド自体)。

一般的な構文

cmdline_arg:string(n:long)

説明

現在のプロセスから要求された引数を返します。それほど多くの引数がない場合や、引数を取得できなかった場合は、空の文字列を返します。通常、引数 0 はコマンド自体です。

名前

function::cmdline_str – 現在のプロセスからすべてのコマンドライン引数を取得します。

概要

function cmdline_str:string()

引数

なし

一般的な構文

cmdline_str:**string**

説明

現在のプロセスから、スペースで区切られたすべての引数を返します。引数を取得できない場合は空の文字列を返します。

名前

function::env_var – 現在のプロセスから環境変数を取得します。

概要

function env_var:string(name:string)

引数

name

取得する環境変数の名前。

一般的な構文

evn_var:string(name:string)

説明
現在のプロセスの指定された環境値の内容を返します。変数が設定されていないと、空の文字列が返されます。

名前
function::print_stack – 文字列からカーネルスタックを出力します。

概要

```
function print_stack(stk:string)
```

引数

stk

16 進アドレスのリストが含まれる文字列。

一般的な構文
print_stack(stk:string)

説明
この関数は、**backtrace** への事前呼び出しの結果となることが仮定される指定の文字列でアドレスのシンボリックルックアップを実行します。

アドレスごとに 1 行出力し、アドレス、アドレスが含まれる関数の名前、およびその関数内での推定位置が含まれます。戻り値はありません。

名前
function::sprint_stack – 文字列からカーネルアドレスのスタックを返します。実験的

概要

```
function sprint_stack:string(stk:string)
```

引数

stk

16 進 (カーネル) アドレスのリストが含まれる文字列。

説明
backtrace への事前呼び出しの結果となることが仮定される指定の文字列でアドレスのシンボリックルックアップを実行します。

指定の 16 進文字列から単純なバックトレースを返します。1 つのアドレスにつき 1 行。シンボル名 (シンボルが解決できない場合は 16 進アドレス) およびモジュール名 (見つかった場合) が含まれます。見つかった場合は、関数の開始点からのオフセットが含まれます。そうでなければ、オフセットはモ

ジュールに追加されます (見つかった場合は括弧間)。文字列としてバックトレースを返します (改行文字で終端される各行)。返されたスタックは MAXSTRINGLEN に切り捨てられます。より完璧で優れたスタックを出力するには、print_stack を使用することに注意してください。

名前
function::probefunc – 既知の場合は、プローブポイントの関数名を返します。

概要

```
function probefunc:string()
```

引数
なし

一般的な構文
probefunc:**string**

説明
この関数は、プローブされている関数の名前を返します。これは、**pp** によって返されるプローブポイント文字列に基づいて行います。

注記
この機能は非推奨になりました。**symname** や **usymname** を使用してください。この関数は、プローブポイントのコンテキストを解析できない場合に、現在のアドレスに基づいて関数名を返す可能性があります。

名前
function::probemod – プローブポイントのカーネルモジュール名を返します。

概要

```
function probemod:string()
```

引数
なし

一般的な構文
probemod:**string**

説明
既知の場合、この関数はプローブポイントが含まれるカーネルモジュールの名前を返します。

名前
function::modname – アドレスでロードされたカーネルモジュール名を返します。

概要

```
function modname:string(addr:long)
```

引数

addr

アドレス。

説明

既知の場合、指定のアドレスに関連付けられたモジュール名を返します。不明な場合には文字列「<unknown>」 が返されます。アドレスがカーネルモジュールではなく、カーネル自体にない場合は、文字列 「kernel」 を返します。

名前

function::symname – 指定のアドレスに関連するカーネルシンボルを返します。

概要

```
function symname:string(addr:long)
```

引数

addr

変換するアドレス。

一般的な構文

symname:string(addr:long)

説明

既知の場合、指定のアドレスに関連する (関数) シンボル名を返します。分からない場合は、addr の 16 進文字列表記を返します。

名前

function::symdata – アドレスのカーネルシンボルとモジュールオフセットを返します。

概要

```
function symdata:string(addr:long)
```

引数

addr

変換するアドレス。

一般的な構文
symdata:string(addr:long)

説明
既知の場合、指定のアドレスに関連する (関数) シンボル名、シンボルの開始およびサイズ、およびモジュール名 (括弧間) を返します。シンボルが不明でもモジュールが分かっている場合は、モジュール内のオフセットとモジュールのサイズが追加されます。要素が不明な場合は省略され、シンボル名が不明な場合は、指定のアドレスの 16 進文字列を返します。

名前
function::usymname – 現在のタスクでのアドレスのシンボルを返します。実験的

概要

```
function usymname:string(addr:long)
```

引数

addr

変換するアドレス。

説明
既知の場合、指定のアドレスに関連する (関数) シンボル名を返します。分からない場合は、addr の 16 進文字列表記を返します。

名前
function::usymdata – アドレスのシンボルとモジュールオフセットを返します。実験的

概要

```
function usymdata:string(addr:long)
```

引数

addr

変換するアドレス。

説明
既知の場合は、現在のタスクの指定のアドレスに関連する (関数) シンボル名、開始からのオフセット、シンボルのサイズ、およびモジュール名 (括弧間) を返します。シンボルが不明でもモジュールが分かっている場合は、モジュール内のオフセットとモジュールのサイズが追加されます。要素が不明な場合は省略され、シンボル名が不明な場合は、指定のアドレスの 16 進文字列を返します。

名前
function::print_ustack – 文字列から現在のタスクのスタックを出力します。実験的

概要

```
function print_ustack(stk:string)
```

引数

stk
現在のタスクの 16 進アドレスのリストが含まれる文字列。

説明
現在のタスクの **ubacktrace** への事前呼び出しの結果となることが仮定される指定の文字列でアドレスのシンボリックルックアップを実行します。

アドレスごとに 1 行出力し、アドレス、アドレスが含まれる関数の名前、およびその関数内での推定位置が含まれます。戻り値はありません。

名前
function::sprint_ustack – 文字列から現在のタスクのスタックを返します。実験的

概要

```
function sprint_ustack:string(stk:string)
```

引数

stk
現在のタスクの 16 進アドレスのリストが含まれる文字列。

説明
現在のタスクの **ubacktrace** への事前呼び出しの結果となることが仮定される指定の文字列でアドレスのシンボリックルックアップを実行します。

指定の 16 進文字列から単純なバックトレースを返します。1 つのアドレスにつき 1 行。シンボル名 (シンボルが解決できない場合は 16 進アドレス) およびモジュール名 (見つかった場合) が含まれます。見つかった場合は、関数の開始点からのオフセットが含まれます。そうでなければ、オフセットはモジュールに追加されます (見つかった場合は括弧間)。文字列としてバックトレースを返します (改行文字で終端される各行)。返されたスタックは MAXSTRINGLEN に切り捨てられます。より完璧で優れたスタックを出力するには、print_ustack を使用することに注意してください。

名前
function::print_backtrace – スタックバックトレースの出力

概要

```
function print_backtrace()
```

引数
なし

一般的な構文
print_backtrace

説明
この関数は print_stack (**backtrace**) と同等ですが、より深いスタックのネストがサポートされます。この関数は値を返しません。

名前
function::sprint_backtrace – スタックバックトレースを文字列として返します。実験的

概要

```
function sprint_backtrace:string()
```

引数
なし

説明
単純な (カーネル) バックトレースを返します。1 つのアドレスにつき 1 行。シンボル名 (シンボルが解決できない場合は 16 進アドレス) およびモジュール名 (見つかった場合) が含まれます。見つかった場合は、関数の開始点からのオフセットが含まれます。そうでなければ、オフセットはモジュールに追加されます (見つかった場合は括弧間)。文字列としてバックトレースを返します (改行文字で終端される各行)。返されたスタックは MAXSTRINGLEN に切り捨てられます。より完璧で優れたスタックを出力するには、**print_backtrace** を使用することに注意してください。Sprint_stack(**backtrace**) と同等ですが、より効率的です (16 進文字列と最終バックトレース文字列間の変換は不要)。

名前
function::backtrace – 現在のスタックの 16 進バックトレース。

概要

```
function backtrace:string()
```

引数
なし

一般的な構文
backtrace:**string**

説明
この関数は、スタックのバックトレースである 16 進アドレスの文字列を返します。出力は文字列の最大長 (MAXSTRINGLEN) に従って切り捨てられることがあります。

名前
function::task_backtrace – 任意タスクの 16 進バックトレース。

概要

```
function task_backtrace:string(task:long)
```

引数

task

task_struct へのポインター

一般的な構文
task_backtrace:string(task:long)

説明
この関数は、特定タスクのスタックのバックトレースである 16 進アドレスの文字列を返します。出力は文字列の最大長に従って切り捨てられることがあります。

名前
function::caller – 呼び出し元関数の名前およびアドレスを返します。

概要

```
function caller:string()
```

引数
なし

一般的な構文
caller:**string**

説明
この関数は呼び出し元関数のアドレスを返します。これは以下の呼び出しに相当します。sprintf(「**s** 0x**x**」, symname(**caller_addr**, **caller_addr**)) 今回は、return プローブにのみ動作します。

名前
function::caller_addr – 呼び出し元アドレスを返します。

概要

```
function caller_addr:long()
```

引数
なし

一般的な構文
caller_addr:**long**

説明
この関数は呼び出し元関数のアドレスを返します。この時点で return プローブに対してのみ動作します。

名前
function::print_ubacktrace – 現在のタスクのスタックバックトレースを出力します。実験的

概要

```
function print_ubacktrace()
```

引数
なし

説明
print_ustack(**ubacktrace**) と同等ですが、より深いスタックのネストがサポートされます。何も返しません。

備考
ユーザー空間アプリケーションと現在のスクリプトに記述されていない共有ライブラリーの (完全な) バックトレースを取得するには、stap を -d /path/to/exe-or-so で実行するか、または --ldd を追加して必要なすべてのアンワインドデータをロードします。

名前
function::sprint_ubacktrace – 現在のタスクのスタックバックトレースを文字列として返します。実験的

概要

```
function sprint_ubacktrace:string()
```

引数
なし

説明
現在のタスクの単純なバックトレースを返します。1 つのアドレスにつき 1 行。シンボル名 (シンボルが解決できない場合は 16 進アドレス) およびモジュール名 (見つかった場合) が含まれます。見つかった

場合は、関数の開始点からのオフセットが含まれます。そうでなければ、オフセットはモジュールに追加されます (見つかった場合は括弧間)。文字列としてバックトレースを返します (改行文字で終端される各行)。返されたスタックは MAXSTRINGLEN に切り捨てられます。より完璧で優れたスタックを出力するには、**print_ubacktrace** を使用することに注意してください。Sprint_ustack(**ubacktrace**) と同等ですが、より効率的です (16 進文字列と最終バックトレース文字列間の変換は不要)。

備考

ユーザー空間アプリケーションと現在のスクリプトに記述されていない共有ライブラリーの (完全な) バックトレースを取得するには、stap を -d /path/to/exe-or-so で実行するか、または --ldd を追加して必要なすべてのアンワインドデータをロードします。

名前

function::print_ubacktrace_brief – 現在のタスクのスタックバックトレースを出力します。実験的

概要

```
function print_ubacktrace_brief()
```

引数

なし

説明

print_ubacktrace と同様ですが、各シンボルの出力は短くなります (名前とオフセットのみ、もしくはシンボルなしの 16 進数アドレスのみ)。

備考

ユーザー空間アプリケーションと現在のスクリプトに記述されていない共有ライブラリーの (完全な) バックトレースを取得するには、stap を -d /path/to/exe-or-so で実行するか、または --ldd を追加して必要なすべてのアンワインドデータをロードします。

名前

function::ubacktrace – 現在のタスクスタックの 16 進バックトレース。実験的

概要

```
function ubacktrace:string()
```

引数

なし

説明

現在のタスクのスタックのバックトレースである 16 進アドレスの文字列を返します。出力は文字列の最大長に従って切り捨てられることがあります。現在のプローブポイントがユーザーのバックトレースを判断できない場合は空の文字列を返します。

備考

ユーザー空間アプリケーションと現在のスクリプトに記述されていない共有ライブラリーの (完全な) バックトレースを取得するには、stap を -d /path/to/exe-or-so で実行するか、または --ldd を追加して必要なすべてのアンワインドデータをロードします。

名前
function::task_current – 現在のタスクの現在の task_struct。

概要

```
function task_current:long()
```

引数
なし

一般的な構文
task_current:**long**

説明
この関数は、現在のプロセスを表す task_struct を返します。このアドレスは、タスク固有のデータを抽出するためにさまざまな task_*() 関数に渡すことができます。

名前
function::task_parent – 親タスクの task_struct。

概要

```
function task_parent:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文
task_parent:long(task:long)

説明
この関数は、指定タスクの親 task_struct を返します。このアドレスは、タスク固有のデータを抽出するためにさまざまな task_*() 関数に渡すことができます。

名前
function::task_state – タスクの状態。

概要

```
function task_state:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文

task_state:long(task:long)

説明

指定タスクの状態を返します (TASK_RUNNING (0)、TASK_INTERRUPTIBLE (1)、TASK_UNINTERRUPTIBLE (2)、TASK_STOPPED (4)、TASK_TRACED (8)、EXIT_ZOMBIE (16)、EXIT_DEAD (32) のいずれか)。

名前

function::task_execname – タスクの名前。

概要

```
function task_execname:string(task:long)
```

引数

task

task_struct ポインター

一般的な構文

task_execname:string(task:long)

説明

指定タスクの名前を返します。

名前

function::task_pid – タスクのプロセス識別子。

概要

```
function task_pid:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文
task_pid:long (task:long)

説明
この関数は、指定タスクのプロセス ID を返します。

名前
function::pid2task – 指定のプロセス識別子の task_struct。

概要

```
function pid2task:long(pid:long)
```

引数

pid
プロセス識別子。

説明
指定のプロセス ID の task 構造を返します。

名前
function::pid2execname – 指定のプロセス識別子の名前。

概要

```
function pid2execname:string(pid:long)
```

引数

pid
プロセス識別子。

説明
指定のプロセス ID の名前を返します。

名前
function::task_tid – タスクのスレッド識別子。

概要

```
function task_tid:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文

task_tid:long(task:long)

説明

この関数は、指定タスクのスレッド ID を返します。

名前

function::task_gid – タスクのグループ識別子。

概要

```
function task_gid:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文

task_gid:long(task:long)

説明

この関数は、指定タスクのグループ ID を返します。

名前

function::task_egid – タスクの実効グループ識別子。

概要

```
function task_egid:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文

task_egid:long(task:long)

説明
この関数は、指定タスクの実効グループ ID を返します。

名前
function::task_uid – タスクのユーザー識別子。

概要

```
function task_uid:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文
task_uid:long(task:long)

説明
この関数は、指定タスクのユーザー ID を返します。

名前
function::task_euid – タスクの実効ユーザー識別子。

概要

```
function task_euid:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文
task_euid:long(task:long)

説明
この関数は、指定タスクの実効ユーザー ID を返します。

名前
function::task_prio – タスクの優先度の値。

概要

```
function task_prio:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文

task_prio:long(task:long)

説明

この関数は、指定タスクの優先度の値を返します。

名前

function::task_nice – タスクの nice 値。

概要

```
function task_nice:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文

task_nice:long(task:long)

説明

この関数は、指定タスクの nice 値を返します。

名前

function::task_cpu – タスクのスケジュールされた CPU。

概要

```
function task_cpu:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文
task_cpu:long(task:long)

説明
この関数は、指定タスクのスケジュールされた CPU を返します。

名前
function::task_open_file_handles – タスクのオープンファイルの数。

概要

```
function task_open_file_handles:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文
task_open_file_handles:long(task:long)

説明
この関数は、指定タスクのオープンファイルハンドラーの数を返します。

名前
function::task_max_file_handles – タスクのオープンファイルの最大数。

概要

```
function task_max_file_handles:long(task:long)
```

引数

task

task_struct ポインター

一般的な構文
task_max_file_handles:long(task:long)

説明
この関数は、指定タスクのファイルハンドラーの最大数を返します。

名前
function::pn – アクティブなプローブ名を返します。

概要

```
function pn:string()
```

引数
なし

一般的な構文
pn:**string**

説明
この関数は、現在実行しているプローブハンドラーに関連するスクリプトレベルのプローブポイントを返します (ワイルドカードによる拡張を含む)。コンテキスト: 現在のプローブポイント。

第4章 タイムスタンプ関数

各タイムスタンプ関数は、関数が実行されるタイミングを示す値を返します。返された値は、イベント発生時、イベントの順序付けの指定、または 2 つのタイムスタンプ間で経過した時間の算出に使用することができます。

名前
function::get_cycles – プロセッサーサイクル数

概要

```
function get_cycles:long()
```

引数
なし

一般的な構文
get_cycles:**long**

説明
この関数はプロセッサーサイクルカウンター値を返します (ある場合)。それ以外の場合はゼロを返します。サイクルカウンターは、各プロセッサーでフリーランで、同期されていない状態です。そのため、異なるプロセッサーの get_cycles 関数の結果を比較することでは、イベントの順序を判断できません。

名前
function::gettimeofday_ns – UNIX エポックからの経過時間 (ナノ秒)

概要

```
function gettimeofday_ns:long()
```

引数
なし

一般的な構文
gettimeofday_ns:**long**

説明
この関数は、UNIX エポックからの経過時間をナノ秒数で返します。

名前
function::gettimeofday_us – UNIX エポックからの経過時間 (マイクロ秒)

概要

function gettimeofday_us:long()

引数
なし

一般的な構文
gettimeofday_us:**long**

説明
この関数は、UNIX エポックからの経過時間をマイクロ秒数で返します。

名前
function::gettimeofday_ms – UNIX エポックからの経過時間 (ミリ秒)

概要

function gettimeofday_ms:long()

引数
なし

一般的な構文
gettimeofday_ms:**long**

説明
この関数は、UNIX エポックからの経過時間をミリ秒数で返します。

名前
function::gettimeofday_s – UNIX エポックからの経過時間 (秒)

概要

function gettimeofday_s:long()

引数
なし

一般的な構文
gettimeofday_s:**long**

説明
この関数は、UNIX エポックからの経過時間を秒数で返します。

第5章 時間文字列ユーティリティー関数

ユーティリティー関数は、タイムスタンプ関数 gettimeofday_s() によって返されるエポックからの経過時間 (秒数) を、人が判読できる日付/時間の文字列に変換します。

名前

function::ctime – エポックからの経過時間 (秒単位) を、人が判読できる日付/時間の文字列に変換します。

概要

```
function ctime:string(epochsecs:long)
```

引数

epochsecs

エポックからの経過時間 (**gettimeofday_s** によって返された場合)。

一般的な構文

ctime:string(epochsecs:long)

説明

gettimeofday_s によって返されるエポックからの経過時間 (秒単位) の引数を取ります。フォームの文字列を返します。

「Wed Jun 30 21:49:08 1993」

文字列は常に 24 文字ちょうどになります。時が異常なほど前である場合 (エポックから秒で、32 ビットオフセットで表現できるもの以前) は、返される文字列は 「a long, long time ago... 」 になります。時が大幅に先の未来である場合、返される文字列は、「far far in the future... 」 になります (これら文字列は両方とも 24 文字)。

エポック (ゼロ) は以下になります。

「Thu Jan 1 00:00:00 1970」

epochsecs -2147483648 に対応する、ctime によって提供される最も初期の日付は、「Fri Dec 13 20:45:52 1901」 です。epochsecs 2147483647 に対応する ctime によって提供される最新の日付は 「Tue Jan 19 03:14:07 2038」 です。

曜日の省略形は 'Sun'、'Mon'、'Tue'、'Wed'、'Thu'、'Fri'、および 'Sat' です。月の省略形は 'Jan'、'Feb'、'Mar'、'Apr'、'Jun'、'Jul'、'Aug'、'Sep'、'Oct'、'Nov'、および 'Dec' です。

実際の C ライブラリー **ctime** 関数は、この関数が文字列の最後に改行文字 (\n) を挿入することに注意してください。また、カーネルにはタイムゾーンの概念がないため、時間は常に GMT で返されます。

第6章 メモリー TAPSET

この種類のプローブポイントは、メモリー関連のイベントをプローブしたり、現在のプロセスのメモリー使用量を照会するために使用されます。これには、以下のプローブポイントが含まれます。

名前

function::vm_fault_contains – ページフォールトの理由の戻り値をテストします。

概要

```
function vm_fault_contains:long(value:long,test:long)
```

引数

value

vm.page_fault.return によって返される fault_type

test

テストするフォールトタイプ (VM_FAULT_OOM または同等)

名前

probe::vm.pagefault – ページフォールトの発生を記録します。

概要

```
vm.pagefault
```

値

write_access

読み取りまたは書き込みアクセスであるかを示します。1 は書き込み、0 は読み取りを示します。

name

プローブポイントの名前。

address

障害が発生しているメモリーアクセスのアドレス (例: ページフォールトの原因となったアドレス)。

コンテキスト

障害を引き起こしたプロセス。

名前

probe::vm.pagefault.return – 発生した障害のタイプを示します。

概要

```
vm.pagefault.return
```

値

name

プローブポイントの名前。

fault_type

メモリー不足による障害は 0 (VM_FAULT_OOM)、小さな障害は 2 (VM_FAULT_MINOR)、大きな障害は 3 (VM_FAULT_MAJOR) を返します。障害がいずれにも該当しない場合は 1 (VM_FAULT_SIGBUS) を返します。

名前

function::addr_to_node – NUMA システム内で指定のアドレスが属するノードを返します。

概要

```
function addr_to_node:long(addr:long)
```

引数

addr

障害が発生しているメモリーアクセスのアドレス

一般的な構文

addr_to_node:long(addr:long)

説明

この関数はアドレスを受け取り、NUMA システム内で指定アドレスが属するノードを返します。

名前

probe::vm.write_shared – 共有ページへの書き込みを試行します。

概要

```
vm.write_shared
```

値

name

プローブポイントの名前。

address
共有書き込みのアドレス。

コンテキスト
コンテキストは、書き込みを試行するプロセスです。

説明
プロセスが共有ページへの書き込みを試みる際に実行されます。コピーが必要な場合は、その後に vm.write_shared_copy が続きます。

名前
probe::vm.write_shared_copy – 共有ページ書き込みのページコピー。

概要

```
vm.write_shared_copy
```

値

name
プローブポイントの名前。

zero
ゼロページであるかどうかを示すブール値 (コピーの代わりに消去を実行可能)。

address
共有書き込みのアドレス。

コンテキスト
書き込みを試行するプロセス。

説明
共有ページへの書き込みでページのコピーが必要な場合に実行されます。これは常に vm.shared_write の前に付けられます。

名前
probe::vm.mmap – mmap が要求されたときに実行されます。

概要

```
vm.mmap
```

値

length

メモリーセグメントの長さ

name

プローブポイントの名前。

address

要求されたアドレス

コンテキスト

mmap を呼び出すプロセス。

名前

probe::vm.munmap – munmap が要求されたときに実行されます。

概要

```
vm.munmap
```

値

length

メモリーセグメントの長さ

name

プローブポイントの名前。

address

要求されたアドレス

コンテキスト

munmap を呼び出すプロセス。

名前

probe::vm.brk – brk が要求されたときに実行されます (例: ヒープのリサイズ)。

概要

```
vm.brk
```

値

length
メモリーセグメントの長さ

name
プローブポイントの名前。

address
要求されたアドレス

コンテキスト
brk を呼び出すプロセス。

名前
probe::vm.oom_kill — OOM Killer によって終了されるスレッドが選択されたときに実行されます。

概要

```
vm.oom_kill
```

値

name
プローブポイントの名前。

task
kill されるタスク。

コンテキスト
余分なメモリーを消費しようとし、OOM を引き起こしたプロセス。

名前
probe::vm.kmalloc — kmalloc が要求されたときに実行されます。

概要

```
vm.kmalloc
```

値

ptr
割り当てられた kmemory へのポインター

caller_function

呼び出し元関数の名前。

call_site

kmemory 関数のアドレス。

gfp_flag_name

割り当てる kmemory のタイプ (文字列形式)。

name

プローブポイントの名前。

bytes_req

要求されたバイト。

bytes_alloc

割り当てられたバイト。

gfp_flags

割り当てる kmemory のタイプ。

名前

probe::vm.kmem_cache_alloc – \ の際に実行されます。

概要

```
vm.kmem_cache_alloc
```

値

ptr

割り当てられた kmemory へのポインター

caller_function

呼び出し元関数の名前。

call_site

この kmemory 関数を呼び出す関数のアドレス。

gfp_flag_name

割り当てる kmemory のタイプ (文字列形式)。

name

プローブポイントの名前。

bytes_req

要求されたバイト。

bytes_alloc
割り当てられたバイト。

gfp_flags
割り当てる kmemory のタイプ。

説明
kmem_cache_alloc が要求されます。

名前
probe::vm.kmalloc_node – kmalloc_node が要求されたときに実行されます。

概要

```
vm.kmalloc_node
```

値

ptr
割り当てられた kmemory へのポインター

caller_function
呼び出し元関数の名前。

call_site
この kmemory 関数を呼び出す関数のアドレス。

gfp_flag_name
割り当てる kmemory のタイプ (文字列形式)。

name
プローブポイントの名前。

bytes_req
要求されたバイト。

bytes_alloc
割り当てられたバイト。

gfp_flags
割り当てる kmemory のタイプ。

名前
probe::vm.kmem_cache_alloc_node – \ の際に実行されます。

概要

```
vm.kmem_cache_alloc_node
```

値

ptr
割り当てられた kmemory へのポインター

caller_function
呼び出し元関数の名前。

call_site
この kmemory 関数を呼び出す関数のアドレス。

gfp_flag_name
割り当てる kmemory のタイプ (文字列形式)。

name
プローブポイントの名前。

bytes_req
要求されたバイト。

bytes_alloc
割り当てられたバイト。

gfp_flags
割り当てる kmemory のタイプ。

説明
kmem_cache_alloc_node が要求されています。

名前
probe::vm.kfree – kfree が要求されたときに実行されます。

概要

```
vm.kfree
```

値

ptr

.

kmalloc によって返された割り当て済み kmemory へのポインター。

caller_function

呼び出し元関数の名前。

call_site

この kmemory 関数を呼び出す関数のアドレス。

name

プローブポイントの名前。

名前

probe::vm.kmem_cache_free – \ の際に実行されます。

概要

```
vm.kmem_cache_free
```

値

ptr

kmem_cache によって返された割り当て済み kmemory へのポインター。

caller_function

呼び出し元関数の名前。

call_site

この kmemory 関数を呼び出す関数のアドレス。

name

プローブポイントの名前。

説明

kmem_cache_free が要求された。

名前

function::proc_mem_size – ページ単位のプログラム仮想メモリーサイズの合計。

概要

```
function proc_mem_size:long()
```

引数
なし

説明
現在のプロセスのページ単位の仮想メモリーサイズ合計を返します。現在のプロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前
function::proc_mem_size_pid – ページ単位のプログラム仮想メモリーサイズの合計。

概要

```
function proc_mem_size_pid:long(pid:long)
```

引数

pid

確認するプロセスの PID。

説明
指定プロセスのページ単位の仮想メモリーサイズ合計を返します。指定のプロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前
function::proc_mem_rss – ページ単位のプログラムレジデント設定サイズ。

概要

```
function proc_mem_rss:long()
```

引数
なし

説明
現在のプロセスのページ単位のレジデント設定サイズを返します。現在のプロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前
function::proc_mem_rss_pid – ページ単位のプログラムレジデント設定サイズ。

概要

```
function proc_mem_rss_pid:long(pid:long)
```

引数

pid

確認するプロセスの PID。

説明

指定プロセスのページ単位のレジデント設定サイズを返します。指定のプロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前

function::proc_mem_shr – プログラム共有ページ (共有マッピングより)。

概要

```
function proc_mem_shr:long()
```

引数

なし

説明

現在のプロセスの共有ページ (共有マッピングより) を返します。現在のプロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前

function::proc_mem_shr_pid – プログラム共有ページ (共有マッピングより)。

概要

```
function proc_mem_shr_pid:long(pid:long)
```

引数

pid

確認するプロセスの PID。

説明

指定プロセスの共有ページ (共有マッピングより) を返します。プロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前

function::proc_mem_txt – ページ単位のプログラムテキスト (コード) サイズ。

概要

function proc_mem_txt:long()

引数
なし

説明
ページ単位の現在のプロセステキスト (コード)サイズを返します。現在のプロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前
function::proc_mem_txt_pid – ページ単位のプログラムテキスト (コード) サイズ。

概要

function proc_mem_txt_pid:long(pid:long)

引数

pid

確認するプロセスの PID。

説明
ページ単位の指定のプロセステキスト (コード) サイズを返します。プロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前
function::proc_mem_data – ページ単位のプログラムデータサイズ (データ + スタック)。

概要

function proc_mem_data:long()

引数
なし

説明
ページ単位の現在のプロセスデータサイズ (データ + スタック) を返します。現在のプロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前
function::proc_mem_data_pid – ページ単位のプログラムデータサイズ (データ + スタック)。

概要

function proc_mem_data_pid:long(pid:long)

引数

pid

確認するプロセスの PID。

説明

ページ単位の指定のプロセスデータサイズ (データ + スタック) を返します。プロセスがない場合や、ページ数を取得できなかった場合はゼロを返します。

名前

function::mem_page_size – このアーキテクチャーのページのバイト数。

概要

function mem_page_size:long()

引数

なし

名前

function::bytes_to_string – 指定バイトの人が判読できる文字列。

概要

function bytes_to_string:string(bytes:long)

引数

bytes

変換するバイト数。

説明

バイト数 (最大 1024 バイト)、'K' が末尾のキロバイト数 (1024K 未満)、'M' が末尾のメガバイト数 (1024M 未満)、または 'G' が末尾のギガバイト数を表す文字列を返します。K、M、または G が付き、数値が 100 未満の場合は、「.」と、その残りが含まれます。返される文字列は 5 文字です (9999G バイトを超える場合や負の値である場合以外は最初に空白文字が挿入されます)。

名前

function::pages_to_string – ページを人が判読できる文字列に変換します。

概要

```
function pages_to_string:string(pages:long)
```

引数

pages

変換するページ数。

説明

ページを **page_size** で乗算してバイト数を取得し、**bytes_to_string** の結果を返します。

名前

function::proc_mem_string – 人が判読できる文字列で示された現在の proc メモリー使用量。

概要

```
function proc_mem_string:string()
```

引数

なし

説明

現在のプロセスによって使用されるメモリーのサイズ、rss、shr、txt、およびデータを表示する、人が判読できる文字列を返します。例: 「size: 301m, rss: 11m, shr: 8m, txt: 52k, data: 2248k 」

名前

function::proc_mem_string_pid – 人が判読できる文字列で示されたプロセスメモリー使用量。

概要

```
function proc_mem_string_pid:string(pid:long)
```

引数

pid

確認するプロセスの PID。

説明

指定のプロセスによって使用されるメモリーのサイズ、rss、shr、txt、およびデータを表示する、人が判読できる文字列を返します。例: 「size: 301m, rss: 11m, shr: 8m, txt: 52k, data: 2248k 」

第7章 タスク時間 TAPSET

この tapset は、ユーティリティー関数を定義して現在のタスクの時間に関するプロパティーを問い合わせ、これらをミリ秒単位で人が判読できる文字列に変換します。

名前
function::task_utime – 現在のタスクのユーザー時間。

概要

```
function task_utime:long()
```

引数
なし

説明
現在のタスクのユーザー時間を cputimeで返します。このプロセスの他のタスクが使用した時間や、このタスクの子が費やした時間は含まれません。

名前
function::task_utime_tid – 指定タスクのユーザー時間。

概要

```
function task_utime_tid:long(tid:long)
```

引数

tid

指定タスクのスレッド ID。

説明
指定タスクのユーザー時間を cputime で返します。タスクが存在しない場合はゼロを返します。このプロセスの他のタスクが使用した時間や、このタスクの子が費やした時間は含まれません。

名前
function::task_stime – 現在のタスクのシステム時間。

概要

```
function task_stime:long()
```

引数
なし

説明

現在のタスクのシステム時間を cputime で返します。このプロセスの他のタスクが使用した時間や、このタスクの子が費やした時間は含まれません。

名前

function::task_stime_tid – 指定タスクのシステム時間。

概要

```
function task_stime_tid:long(tid:long)
```

引数

tid

指定タスクのスレッド ID。

説明

指定タスクのシステム時間を cputime で返します。タスクが存在しない場合はゼロを返します。このプロセスの他のタスクが使用した時間や、このタスクの子が費やした時間は含まれません。

名前

function::cputime_to_msecs – 指定の CPU 時間をミリ秒に変換します。

概要

```
function cputime_to_msecs:long(cputime:long)
```

引数

cputime

ミリ秒に変換する時間。

名前

function::msecs_to_string – 人が判読できる文字列で示された指定のミリ秒。

概要

```
function msecs_to_string:string(msecs:long)
```

引数

msecs

変換するミリ秒数。

説明

ミリ秒数を人が判読できる文字列で返します。この文字列の形式は 「XmY.ZZZs」 で、X は分数、Y は秒数、ZZZ はミリ秒数を示します。

名前

function::cputime_to_string – 人が判読できる文字列で示された指定の CPU 時間。

概要

```
function cputime_to_string:string(cputime:long)
```

引数

cputime

変換する時間。

説明

msec_to_string (cputime_to_msecs (cputime) の呼び出しと同等です。

名前

function::task_time_string – 人が判読できる文字列で示されたタスク時間の使用量。

概要

```
function task_time_string:string()
```

引数

なし

説明

現在のタスクがこれまでに使用したユーザーおよびシステム時間を示す、人が判読可能な文字列を返します。例: 「usr: 0m12.908s, sys: 1m6.851s」

名前

function::task_time_string_tid – 人が判読できる文字列で示されたタスク時間の使用量。

概要

```
function task_time_string_tid:string(tid:long)
```

引数

tid

指定タスクのスレッド ID。

説明

指定のタスクがこれまでに使用したユーザーおよびシステム時間を示す、人が判読可能な文字列を返します。例: 「usr: 0m12.908s, sys: 1m6.851s」

第8章 IO スケジューラーおよびブロック IO TAPSET

この種類のプローブポイントは、ブロック IO レイヤーおよび IO スケジューラーの活動をプローブするために使用されます。以下のプローブポイントが含まれます。

名前

probe::ioscheduler.elv_next_request – 要求キューから要求が取得されたときに実行されます。

概要

```
ioscheduler.elv_next_request
```

値

name

プローブポイントの名前。

elevator_name

現在有効になっている I/O エレベーターのタイプ。

名前

probe::ioscheduler.elv_next_request.return – 要求の取得によってシグナルが返されると実行されます。

概要

```
ioscheduler.elv_next_request.return
```

値

disk_major

要求のディスクメジャー番号。

rq

要求のアドレス。

name

プローブポイントの名前。

disk_minor

要求のディスクマイナー番号。

rq_flags

要求フラグ。

名前

probe::ioscheduler.elv_completed_request – 要求が完了すると実行されます

概要

```
ioscheduler.elv_completed_request
```

値

disk_major

要求のディスクメジャー番号。

rq

要求のアドレス。

name

プローブポイントの名前。

elevator_name

現在有効になっている I/O エレベーターのタイプ。

disk_minor

要求のディスクマイナー番号。

rq_flags

要求フラグ。

名前

probe::ioscheduler.elv_add_request.kp – 要求が要求キューに追加されたことを示す kprobe ベースのプローブ。

概要

```
ioscheduler.elv_add_request.kp
```

値

disk_major

要求のディスクメジャー番号。

rq

要求のアドレス。

q

要求キューへのポインター。

name
プローブポイントの名前。

elevator_name
現在有効になっている I/O エレベーターのタイプ。

disk_minor
要求のディスクマイナー番号。

rq_flags
要求フラグ。

名前
probe::ioscheduler.elv_add_request.tp – 要求が要求キューに追加されたことを示す tracepoint ベースのプローブ。

概要

```
ioscheduler.elv_add_request.tp
```

値

disk_major
要求のディスクメジャー番号。

rq
要求のアドレス。

q
要求キューのポインター。

name
プローブポイントの名前。

elevator_name
現在有効になっている I/O エレベーターのタイプ。

disk_minor
要求のディスクマイナー番号。

rq_flags
要求フラグ。

名前

probe::ioscheduler.elv_add_request – 要求が要求キューに追加されたことを示すプローブ。

概要

```
ioscheduler.elv_add_request
```

値

disk_major

要求のディスクメジャー番号。

rq

要求のアドレス。

q

要求キューのポインター。

elevator_name

現在有効になっている I/O エレベーターのタイプ。

disk_minor

要求のディスクマイナー番号。

rq_flags

要求フラグ。

名前

probe::ioscheduler_trace.elv_completed_request – 要求が完了すると実行されます。

概要

```
ioscheduler_trace.elv_completed_request
```

値

disk_major

要求のディスクメジャー番号。

rq

要求のアドレス。

name

プローブポイントの名前。

elevator_name
現在有効になっている I/O エレベーターのタイプ。

disk_minor
要求のディスクマイナー番号。

rq_flags
要求フラグ。

説明
完了済み。

名前
probe::ioscheduler_trace.elv_issue_request – 要求が完了すると実行されます。

概要

```
ioscheduler_trace.elv_issue_request
```

値

disk_major
要求のディスクメジャー番号。

rq
要求のアドレス。

name
プローブポイントの名前。

elevator_name
現在有効になっている I/O エレベーターのタイプ。

disk_minor
要求のディスクマイナー番号。

rq_flags
要求フラグ。

説明
スケジュール済み。

名前
probe::ioscheduler_trace.elv_requeue_request – 要求が完了すると実行されます。

概要

```
ioscheduler_trace.elv_requeue_request
```

値

disk_major
要求のディスクメジャー番号。

rq
要求のアドレス。

name
プローブポイントの名前。

elevator_name
現在有効になっている I/O エレベーターのタイプ。

disk_minor
要求のディスクマイナー番号。

rq_flags
要求フラグ。

説明
ハードウェアがこれ以上要求を受け入れることができないときにキューに戻されます。

名前
probe::ioscheduler_trace.elv_abort_request – 要求が中止されると実行されます。

概要

```
ioscheduler_trace.elv_abort_request
```

値

disk_major
要求のディスクメジャー番号。

rq
要求のアドレス。

name
プローブポイントの名前。

elevator_name
現在有効になっている I/O エレベーターのタイプ。

disk_minor
要求のディスクマイナー番号。

rq_flags
要求フラグ。

名前
probe::ioscheduler_trace.plug – 要求キューがプラグされると実行されます。

概要

```
ioscheduler_trace.plug
```

値

name
プローブポイントの名前。

rq_queue
要求キュー

説明
キュー内の要求はブロックドライバーで処理できません。

名前
probe::ioscheduler_trace.unplug_io – 要求キューがアンプラグされると実行されます。

概要

```
ioscheduler_trace.unplug_io
```

値

name
プローブポイントの名前。

rq_queue

要求キュー

説明
キュー内の保留中の要求の数がしきい値を超えたとき、またはキューがプラグされたときにアクティブ化されたタイマーの終了時。

名前
probe::ioscheduler_trace.unplug_timer – アンプラグタイマーが関連付けられたときに実行されます。

概要

```
ioscheduler_trace.unplug_timer
```

値

name

プローブポイントの名前。

rq_queue

要求キュー

説明
要求キューの終了時。

名前
probe::ioblock.request – 汎用的なブロック I/O 要求を行ったときに必ず実行されます。

概要

```
ioblock.request
```

値
なし

説明
name: プローブポイントの名前 ***devname***: ブロックデバイス名 ***ino***: マップされたファイルの i-node 番号 ***sector***: bio 全体のセクターの開始セクター ***flags***: 以下を参照。BIO_UPTODATE 0 I/O 完了後 ok、BIO_RW_BLOCK 1 RW_AHEAD セット、読み取り/書き込みのブロック、BIO_EOF 2 out-out-bounds エラー、BIO_SEG_VALID 3 nr_hw_seg が有効、BIO_CLONED 4 データを所有しない、BIO_BOUNCED 5 bio はバウンス bio、BIO_USER_MAPPED 6 ユーザーページを含む、BIO_EOPNOTSUPP 7 サポートされない。

rw: 読み書き要求のバイナリートレース ***vcnt***: この I/O 要求を構成するアレイ要素 (ページ、オフセット、長さ) の数を表す ***idx***: bio ベクターアレイへのオフセット ***phys_segments***: 物理アドレスコアレッシングが実行された後のこの bio のセグメント数 ***hw_segments***: 物理および DMA 再マッピングハード

ウェアコアレッシングが実行された後のセグメントの数 ***size***: 合計バイト数 ***bdev***: ターゲットブロックデバイス ***bdev_contains***: パーティションを含むデバイスオブジェクトを参照 (bio 構造がパーティションを表す場合) ***p_start_sect*** デバイスのパーティション構造の開始セクターを参照

コンテキスト
ブロック I/O 要求を行うプロセス

名前
probe::ioblock.end – ブロック I/O 転送が完了したときに必ず実行されます。

概要

```
ioblock.end
```

値
なし

説明
name: プローブポイントの名前 ***devname***: ブロックデバイス名 ***ino***: マップされたファイルの i-node 番号 ***bytes_done***: 転送済みバイト数 ***sector***: bio 全体の開始セクター ***flags***: 以下を参照。BIO_UPTODATE 0 I/O 完了後 ok、BIO_RW_BLOCK 1 RW_AHEAD セット、読み取り/書き込みのブロック、BIO_EOF 2 out-out-bounds エラー、BIO_SEG_VALID 3 nr_hw_seg が有効、BIO_CLONED 4 データを所有しない、BIO_BOUNCED 5 bio はバウンス bio、BIO_USER_MAPPED 6 ユーザーページを含む、BIO_EOPNOTSUPP 7 サポートされない。***error*** : 読み取り/書き込み要求のバイナリートレースの成功で 0 ***rw***: この I/O 要求を形成するアレイ要素 (ページ、オフセット、長さ) の数 を表す bio ベクターカウント ***vcnt*** - 読み取り/書き込み要求のバイナリートレース ***idx***: bio ベクターアレイへのオフセット ***phys_segments***: 物理および DMA 再マッピングハードウェアコアレッシングが実行された後のセグメントの数***hw_segments***: 物理および DMA 再マッピングハードウェアコアレッシングが実行された後のセグメントの数***size***: 合計バイトサイズ

コンテキスト
プロセスは転送が行われたことを示します。

名前
probe::ioblock_trace.bounce – ブロック I/O 要求の少なくとも 1 つのページに対してバッファーバウンスが必要なときに必ず実行されます。

概要

```
ioblock_trace.bounce
```

値
なし

説明
name: プローブポイントの名前 ***q*** - この bio がキューに格納された要求キュー ***devname***: バッファーバウンスが必要とされたデバイス ***ino***: マッピングされたファイルの i-node 番号 ***bytes_done*** - 転送済みのバイト数 ***sector***: bio 全体の開始セクター ***flags***: 以下を参照。BIO_UPTODATE 0 I/O 完了後 ok、

BIO_RW_BLOCK 1 RW_AHEAD セット、読み取り/書き込みのブロック、BIO_EOF 2 out-out-bounds エラー、BIO_SEG_VALID 3 nr_hw_seg が有効、BIO_CLONED 4 データを所有しない、BIO_BOUNCED 5 bio はバウンス bio、BIO_USER_MAPPED 6 ユーザーページを含む、BIO_EOPNOTSUPP 7 サポートされない。***rw***: 読み書き要求のバイナリートレース ***vcnt***: この I/O 要求を形成するアレイ要素 (ページ、オフセット、長さ) の数を表す bio ベクターカウント ***idx***: bio ベクターアレイへのオフセット ***phys_segments***: 物理アドレスコアレッシングが実行された後のこの bio のセグメント数 ***size***: 合計バイト数 ***bdev***: ターゲットブロックデバイス ***bdev_contains***: パーティションを含むデバイスオブジェクトを参照します (bio 構造がパーティションを表す場合) ***p_start_sect***: 出デバイスのパーティション構造の開始セクターを参照

コンテキスト
ブロック I/O 要求を作成するプロセス。

名前
probe::ioblock_trace.request – bio に対して汎用的なブロック I/O 要求が作成されるときに実行されます。

概要

```
ioblock_trace.request
```

値
なし

説明
name: プローブポイントの名前 ***q***: この bio がキューに格納された要求キュー :ブロックデバイス名 ***devname***:ブロックデバイス名 ***ino***: マップされたファイルの i-node 番号 ***bytes_done***: 転送済みのバイト数 ***sector***: bio 全体のセクターの開始セクター ***flags***: 以下を参照。BIO_UPTODATE 0 I/O 完了後 ok、BIO_RW_BLOCK 1 RW_AHEAD セット、読み取り/書き込みのブロック、BIO_EOF 2 out-out-bounds エラー、BIO_SEG_VALID 3 nr_hw_seg が有効、BIO_CLONED 4 データを所有しない、BIO_BOUNCED 5 bio はバウンス bio、BIO_USER_MAPPED 6 ユーザーページを含む、BIO_EOPNOTSUPP 7 サポートされない。

rw: 読み書き要求のバイナリートレース ***vcnt***: この I/O 要求を構成するアレイ要素 (ページ、オフセット、長さ) の数を表す ***idx***: bio ベクターアレイへのオフセット ***phys_segments***: 物理アドレスコアレッシングが実行された後のこの bio のセグメント数 ***size***: 合計バイト数 ***bdev***: ターゲットブロックデバイス ***bdev_contains***: パーティションを含むデバイスオブジェクトを参照 (bio 構造がパーティションを表す場合) ***p_start_sect***: デバイスのパーティション構造の開始セクターを参照

コンテキスト
ブロック I/O 要求を行うプロセス

名前
probe::ioblock_trace.end – ブロック I/O 転送が完了したときに必ず実行されます。

概要

```
ioblock_trace.end
```

値
なし

説明
name: プローブポイントの名前 ***q***: この bio がキューに格納された要求キュー :ブロックデバイス名 ***devname***:ブロックデバイス名 ***ino***: マップされたファイルの i-node 番号 ***bytes_done***: 転送済みのバイト数 ***sector***: bio 全体のセクターの開始セクター ***flags***: 以下を参照。BIO_UPTODATE 0 I/O 完了後 ok、BIO_RW_BLOCK 1 RW_AHEAD セット、読み取り/書き込みのブロック、BIO_EOF 2 out-out-bounds エラー、BIO_SEG_VALID 3 nr_hw_seg が有効、BIO_CLONED 4 データを所有しない、BIO_BOUNCED 5 bio はバウンス bio、BIO_USER_MAPPED 6 ユーザーページを含む、BIO_EOPNOTSUPP 7 サポートされない。

rw: 読み書き要求のバイナリートレース ***vcnt***: この I/O 要求を構成するアレイ要素 (ページ、オフセット、長さ) の数を表す ***idx***: bio ベクターアレイへのオフセット ***phys_segments***: 物理アドレスコアレッシングが実行された後のこの bio のセグメント数 ***size***: 合計バイト数 ***bdev***: ターゲットブロックデバイス ***bdev_contains***: パーティションを含むデバイスオブジェクトを参照 (bio 構造がパーティションを表す場合) ***p_start_sect***: デバイスのパーティション構造の開始セクターを参照

コンテキスト
プロセスは転送が行われたことを示します。

第9章 SCSI TAPSET

この種類のプローブポイントは、SCSI アクティビティーをプローブするために使用されます。以下のプローブポイントが含まれます。

名前

probe::scsi.ioentry – SCSI 中間層要求の作成

概要

```
scsi.ioentry
```

値

disk_major

ディスクのメジャー番号 (未指定の場合は -1)

device_state_str

デバイスの現在の状態 (文字列)

device_state

デバイスの現在の状態

req_addr

現在の構造要求ポインター (数字)

disk_minor

ディスクのマイナー番号 (未指定の場合は -1)

名前

probe::scsi.iodispatching – SCSI 中間層でディスパッチされるローレベル SCSI コマンド

概要

```
scsi.iodispatching
```

値

device_state_str

デバイスの現在の状態 (文字列)

dev_id

SCSI デバイス ID

channel

チャネル番号

data_direction

data_direction は、このコマンドがデバイス 0 (DMA_BIDIRECTIONAL)、1 (DMA_TO_DEVICE)、2 (DMA_FROM_DEVICE)、3 (DMA_NONE) から送信されるか、デバイスに送信されるかを指定します。

lun

LUN 番号

request_bufflen

要求バッファーの長さ

host_no

ホスト番号

device_state

デバイスの現在の状態

data_direction_str

データの方向 (文字列)

req_addr

現在の構造要求ポインター (数字)

request_buffer

要求バッファーアドレス

名前

probe::scsi.iodone – 低レベルドライバーによって完了され、実行済みキューに格納される SCSI コマンド。

概要

```
scsi.iodone
```

値

device_state_str

デバイスの現在の状態 (文字列)

dev_id

SCSI デバイス ID

channel

チャネル番号

data_direction

data_direction は、このコマンドがデバイスから送信されるか、デバイスに送信されるかを指定します。

lun

LUN 番号

host_no

ホスト番号

data_direction_str

データの方向 (文字列)

device_state

デバイスの現在の状態

scsi_timer_pending

タイマーがこの要求で保留中の場合は 1

req_addr

現在の構造要求ポインター (数字)

名前

probe::scsi.iocompleted – ブロックデバイス I/O 要求の完了処理を行っている SCSI 中間層

概要

```
scsi.iocompleted
```

値

device_state_str

デバイスの現在の状態 (文字列)

dev_id

SCSI デバイス ID

channel

チャネル番号

data_direction

data_direction は、このコマンドがデバイスから送信されるか、デバイスに送信されるかを指定します。

lun
LUN 番号

host_no
ホスト番号

data_direction_str
データの方向 (文字列)

device_state
デバイスの現在の状態

req_addr
現在の構造要求ポインター (数字)

goodbytes
完了済みバイト数

名前
probe::scsi.ioexecute – 中間層 SCSI 要求の作成および結果の待機

概要

```
scsi.ioexecute
```

値

retries
要求を再試行する回数

device_state_str
デバイスの現在の状態 (文字列)

dev_id
SCSI デバイス ID

channel
チャネル番号

data_direction
data_direction は、このコマンドがデバイスから送信されるか、デバイスに送信されるかを指定します。

lun
LUN 番号

timeout
要求タイムアウト (秒単位)

request_bufflen
データバッファーの長さ

host_no
ホスト番号

data_direction_str
データの方向 (文字列)

device_state
デバイスの現在の状態

request_buffer
データバッファーアドレス

名前
probe::scsi.set_state – SCSI デバイス状態の変更を要求します。

概要

```
scsi.set_state
```

値

state_str
デバイスの新しい状態 (文字列)

dev_id
SCSI デバイス ID

channel
チャネル番号

state
デバイスの新しい状態

old_state_str
デバイスの現在の状態 (文字列)

lun
LUN 番号

old_state

デバイスの現在の状態

host_no

ホスト番号

第10章 TTY TAPSET

この種類のプローブポイントは、TTY(Teletype) のアクティビティーをプローブするために使用されます。以下のプローブポイントが含まれます。

名前

probe::tty.open – tty が開かれると呼び出されます。

概要

```
tty.open
```

値

inode_state

inode 状態

file_name

ファイル名

file_mode

ファイルモード

file_flags

ファイルフラグ

inode_number

inode 番号

inode_flags

inode フラグ

名前

probe::tty.release – tty が閉じると呼び出されます。

概要

```
tty.release
```

値

inode_state

inode 状態

file_name

ファイル名

file_mode
ファイルモード

file_flags
ファイルフラグ

inode_number
inode 番号

inode_flags
inode フラグ

名前
probe::tty.resize – 端末のサイズが変更される際に呼び出されます。

概要

```
tty.resize
```

値

new_ypixel
新規の ypixel 値

old_col
古い列の値

old_xpixel
古い xpixel

old_ypixel
古い ypixel

name
tty 名

old_row
古い行の値

new_row
新規の行の値

new_xpixel

新規の xpixel 値

new_col

新規の列の値

名前

probe::tty.ioctl – ioctl が tty に要求される際に呼び出されます。

概要

```
tty.ioctl
```

値

cmd

ioctl コマンド

arg

ioctl 引数

name

ファイル名

名前

probe::tty.init – tty が初期化される際に呼び出されます。

概要

```
tty.init
```

値

driver_name

ドライバー名

name

ドライバー .dev_name の名前

module

モジュール名

名前

probe::tty.register – tty デバイスが登録されると呼び出されます。

概要

```
tty.register
```

値

driver_name

ドライバー名

name

ドライバー .dev_name の名前

index

要求される tty インデックスです。

module

モジュール名

名前

probe::tty.unregister – tty デバイスの登録が解除されると呼び出されます。

概要

```
tty.unregister
```

値

driver_name

ドライバー名

name

ドライバー .dev_name の名前

index

要求される tty インデックスです。

module

モジュール名

名前
probe::tty.poll – tty デバイスがポーリングされる際に呼び出されます。

概要

```
tty.poll
```

値

file_name
tty ファイル名

wait_key
待機キューキー

名前
probe::tty.receive – tty がメッセージを受信すると呼び出されます。

概要

```
tty.receive
```

値

driver_name
ドライバー名

count
受信する文字数

name
モジュールファイルの名前

fp
フラグバッファー

cp
受信したバッファー

index
tty インデックス

id
tty id

名前
probe::tty.write – tty 行に書き込みます。

概要

```
tty.write
```

値

driver_name
ドライバー名

buffer
書き込まれるバッファー

file_name
tty に関連するファイル名

nr
文字数

名前
probe::tty.read – tty 行が読み込まれる際に呼び出されます。

概要

```
tty.read
```

値

driver_name
ドライバー名

buffer
文字を受信するバッファー

file_name
tty に関連するファイル名

nr
読み込まれる文字数

第11章 ネットワーキング TAPSET

この種類のプローブポイントは、ネットワークデバイスおよびプロトコル層の活動をプローブするために使用されます。

名前

probe::netdev.receive – ネットワークデバイスから受信されたデータ

概要

```
netdev.receive
```

値

protocol

受信されるパケットのプロトコル。

dev_name

デバイスの名前 (例: eth0、ath1)。

length

受信バッファーの長さ。

名前

probe::netdev.transmit – ネットワークデバイスの送信バッファー

概要

```
netdev.transmit
```

値

protocol

このパケットのプロトコル (include/linux/if_ether.h で定義されます)。

dev_name

デバイスの名前 (例: eth0、ath1)。

length

送信バッファーの長さ。

truesize

送信されるデータのサイズ

名前
probe::netdev.change_mtu – netdev MTU が変更されたときに呼び出されます。

概要

```
netdev.change_mtu
```

値

dev_name
MTU が変更されたデバイス

new_mtu
新しい MTU

old_mtu
現在の MTU

名前
probe::netdev.open – デバイスが開いたときに呼び出されます。

概要

```
netdev.open
```

値

dev_name
開くデバイス

名前
probe::netdev.close – デバイスが閉じられたときに呼び出されます。

概要

```
netdev.close
```

値

dev_name
閉じられるデバイス

名前
probe::netdev.hard_transmit – デバイスが TX (ハード) に移動するときに呼び出されます。

概要

```
netdev.hard_transmit
```

値

protocol
送信で使用されるプロトコル

dev_name
送信するようスケジュールされたデバイス

length
送信バッファーの長さ。

truesize
送信されるデータのサイズ

名前
probe::netdev.rx – デバイスがパケットを受信するときに呼び出されます。

概要

```
netdev.rx
```

値

protocol
パケットプロトコル

dev_name
パケットを受信したデバイス。

名前
probe::netdev.change_rx_flag – デバイスの RX フラグが変更されたときに呼び出されます。

概要

```
netdev.change_rx_flag
```

値

dev_name

変更されるデバイス

flags

新しいフラグ

名前

probe::netdev.set_promiscuity – デバイスのプロミスキャスモードが開始されたとき、またはプロミスキャスモードが終了したときに呼び出されます。

概要

```
netdev.set_promiscuity
```

値

dev_name

プロミスキャスモードが開始される、またはプロミスキャスモードが終了するデバイス。

enable

デバイスでプロミスキャスモードが開始される場合

inc

プロミスキャスモードオープナーの数をカウントします。

disable

デバイスが promiscuity モードを脱退する場合

名前

probe::netdev.ioctl – デバイスで IOCTL が発生したときに呼び出されます。

概要

```
netdev.ioctl
```

値

cmd

IOCTL 要求

arg

IOCTL 引数 (通常は netdev インターフェース)

名前
probe::netdev.register – デバイスが登録されたときに呼び出されます。

概要

```
netdev.register
```

値

dev_name
登録されるデバイス

名前
probe::netdev.unregister – デバイスの登録が解除されると呼び出されます。

概要

```
netdev.unregister
```

値

dev_name
登録解除されるデバイス

名前
probe::netdev.get_stats – デバイス統計の問い合わせが行われるときに呼び出されます。

概要

```
netdev.get_stats
```

値

dev_name
統計を提供するデバイス

名前
probe::netdev.change_mac – netdev_name により MAC が変更されたときに呼び出されます。

概要

```
netdev.change_mac
```

値

dev_name
MTU が変更されたデバイス

new_mac
新しい MAC アドレス

mac_len
MAC 長

old_mac
現在の MAC アドレス

名前
probe::tcp.sendmsg – TCP メッセージの送信。

概要

```
tcp.sendmsg
```

値

name
このプローブの名前。

size
送信バイト数。

sock
ネットワークソケット。

コンテキスト
TCP メッセージを送信するプロセス。

名前

probe::tcp.sendmsg.return – TCP メッセージの送信が行われました。

概要

```
tcp.sendmsg.return
```

値

name

このプローブの名前。

size

エラーが発生した場合の送信バイト数またはエラーコード。

コンテキスト

TCP メッセージを送信するプロセス。

名前

probe::tcp.recvmsg – TCP メッセージの受信。

概要

```
tcp.recvmsg
```

値

saddr

ソース IP アドレスを表す文字列

daddr

宛先 IP アドレスを表す文字列

name

このプローブの名前。

sport

TCP ソースポート。

dport

TCP 宛先ポート。

size

受信バイト数。

sock

ネットワークソケット。

コンテキスト

TCP メッセージを受信するプロセス。

名前

probe::tcp.recvmsg.return – TCP メッセージの受信が完了しました。

概要

```
tcp.recvmsg.return
```

値

saddr

ソース IP アドレスを表す文字列

daddr

宛先 IP アドレスを表す文字列

name

このプローブの名前。

sport

TCP ソースポート。

dport

TCP 宛先ポート。

size

エラーが発生した場合の受信バイト数またはエラーコード。

コンテキスト

TCP メッセージを受信するプロセス。

名前

probe::tcp.disconnect – TCP ソケットの接続解除

概要

```
tcp.disconnect
```

値

saddr
ソース IP アドレスを表す文字列

daddr
宛先 IP アドレスを表す文字列

flags
TCP フラグ (FIN など)

name
このプローブの名前。

sport
TCP ソースポート。

dport
TCP 宛先ポート。

sock
ネットワークソケット。

コンテキスト
TCP を接続解除するプロセス

名前
probe::tcp.disconnect.return – TCP ソケットの接続解除が完了しました。

概要

```
tcp.disconnect.return
```

値

ret
エラーコード (0: エラーなし)

name
このプローブの名前。

コンテキスト
TCP を接続解除するプロセス

名前
probe::tcp.setsockopt – **setsockopt** に呼び出します

概要

```
tcp.setsockopt
```

値

optstr
optname を人間が判読できる形式に解決します。

level
ソケットオプションが処理されるレベル。

optlen
setsockopt の値のアクセスに使用

name
このプローブの名前。

optname
TCP ソケットオプション (TCP_NODELAY や TCP_MAXSEG など)

sock
ネットワークソケット。

コンテキスト
setsockopt を呼び出すプロセス。

名前
probe::tcp.setsockopt.return – **setsockopt** からの戻り値

概要

```
tcp.setsockopt.return
```

値

ret
エラーコード (0: エラーなし)

name
このプローブの名前。

コンテキスト
setsockopt を呼び出すプロセス。

名前
probe::tcp.receive – TCP パケットが受信されたときに呼び出されます。

概要

```
tcp.receive
```

値

urg
TCP URG フラグ

protocol
ドライバーからのパケットプロトコル

psh
TCP PSH フラグ

name
プローブポイントの名前。

rst
TCP RST フラグ

dport
TCP 宛先ポート。

saddr
ソース IP アドレスを表す文字列

daddr
宛先 IP アドレスを表す文字列

ack
TCP ACK フラグ

fin
TCP FIN フラグ

syn
TCP SYN フラグ

sport
TCP ソースポート。

iphdr
IP ヘッダーアドレス

名前
probe::udp.sendmsg – プロセスが UDP メッセージを送信するときに必ず実行されます。

概要

```
udp.sendmsg
```

値

name
このプローブの名前

size
プロセスが送信したバイト数

sock
プロセスにより使用されるネットワークソケット

コンテキスト
UDP メッセージを送信したプロセス

名前
probe::udp.sendmsg.return – UDP メッセージの送信試行が完了したときに必ず実行されます

概要

```
udp.sendmsg.return
```

値

name
このプローブの名前

size
プロセスが送信したバイト数

コンテキスト
UDP メッセージを送信したプロセス

名前
probe::udp.recvmsg – UDP メッセージが受信されたときに必ず実行されます

概要

```
udp.recvmsg
```

値

name
このプローブの名前

size
プロセスが受信したバイト数

sock
プロセスにより使用されるネットワークソケット

コンテキスト
UDP メッセージを受信したプロセス

名前
probe::udp.recvmsg.return – UDP メッセージの受信試行が完了したときに必ず実行されます

概要

```
udp.recvmsg.return
```

値

name
このプローブの名前

size
プロセスが受信したバイト数

コンテキスト
UDP メッセージを受信したプロセス

名前
probe::udp.disconnect – プロセスが UDP の接続解除を要求したときに実行されます。

概要

```
udp.disconnect
```

値

flags
フラグ (FIN など)

name
このプローブの名前

sock
プロセスにより使用されるネットワークソケット

コンテキスト
UDP の接続解除を要求するプロセス

名前
probe::udp.disconnect.return – UDP が正常に接続解除されました。

概要

```
udp.disconnect.return
```

値

ret
エラーコード (0: エラーなし)

name
このプローブの名前

コンテキスト
UDP の接続解除を要求したプロセス

名前
function::ip_ntop – 整数の IP 番号から表現した文字列を返します。

概要

```
-
```

function ip_ntop:string(addr:long)

引数

addr

整数で表される ip

第12章 ソケット TAPSET

この種類のプローブポイントは、ソケットの活動をプローブするために使用されます。以下のプローブポイントが含まれます。

名前

probe::socket.send – ソケットで送信されたメッセージ。

概要

```
socket.send
```

値

success

正常に送信されたか？(1 = yes、0 = no)

protocol

プロトコル値

flags

ソケットフラグ値

name

このプローブの名前。

state

ソケット状態値

size

送信されるメッセージのサイズ (バイト単位) またはエラーコード (success = 0 の場合)

type

ソケットタイプの値。

family

プロトコルファミリー値

コンテキスト

メッセージの送信者

名前

probe::socket.receive – ソケットで受信されたメッセージ

概要

```
socket.receive
```

値

success

正常に送信されたか？(1 = yes、0 = no)

protocol

プロトコル値

flags

ソケットフラグ値

name

このプローブの名前。

state

ソケット状態値

size

受信されるメッセージのサイズ (バイト単位) またはエラーコード (success = 0 の場合)

type

ソケットタイプの値。

family

プロトコルファミリー値

コンテキスト

メッセージ受信者

名前

probe::socket.sendmsg – メッセージをソケットで送信中です。

概要

```
socket.sendmsg
```

値

protocol

プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
メッセージサイズ (バイト単位)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージの送信者

説明
sock_sendmsg 関数から、ソケットでのメッセージ送信の開始時に実行されます。

名前
probe::socket.sendmsg.return – socket.sendmsg から返します。

概要

```
socket.sendmsg.return
```

値

success
正常に送信されたか？(1 = yes、0 = no)

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
送信されるメッセージのサイズ (バイト単位) またはエラーコード (success = 0 の場合)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージの送信者。

説明
sock_sendmsg 関数から、ソケットでのメッセージ送信の終了時に実行されます。

名前
probe::socket.recvmsg – メッセージをソケットで受信中です。

概要

```
socket.recvmsg
```

値

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
メッセージサイズ (バイト単位)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージ受信者。

説明
sock_recvmsg 関数からの、ソケットでのメッセージ受信の開始時に実行されます。

名前
probe::socket.recvmsg.return – ソケットで受信中のメッセージから返します

概要

```
socket.recvmsg.return
```

値

success
正常に受信されたかどうか (1 = yes、0 = no)。(1 = yes、0 = no)

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
受信されるメッセージのサイズ (バイト単位) またはエラーコード (success = 0 の場合)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージ受信者。

説明
sock_recvmsg 関数からの、ソケットでのメッセージ受信の終了時に実行されます。

名前
probe::socket.aio_write – メッセージは **sock_aio_write** から送信されます

概要

```
socket.aio_write
```

値

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
メッセージサイズ (バイト単位)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージの送信者

説明
sock_aio_write 関数から、ソケットでのメッセージ送信の開始時に実行されます。

名前
probe::socket.aio_write.return – **sock_aio_write** からメッセージの結論を送信

概要

■

socket.aio_write.return

値

success
正常に受信されたかどうか (1 = yes、0 = no)。(1 = yes、0 = no)

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
受信されるメッセージのサイズ (バイト単位) またはエラーコード (success = 0 の場合)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージ受信者。

説明
sock_aio_write 関数から、ソケットでのメッセージ送信の終了時に実行されます。

名前
probe::socket.aio_read – **sock_aio_read** からのメッセージの受信

概要

socket.aio_read

値

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
メッセージサイズ (バイト単位)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージの送信者

説明
sock_aio_read 関数からの、ソケットでのメッセージ受信の開始時に実行されます。

名前
probe::socket.aio_read.return – **sock_aio_read** から受信したメッセージの結論

概要

```
socket.aio_read.return
```

値

success
正常に受信されたかどうか (1 = yes、0 = no)。(1 = yes、0 = no)

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
受信されるメッセージのサイズ (バイト単位) またはエラーコード (success = 0 の場合)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージ受信者。

説明
sock_aio_read 関数からの、ソケットでのメッセージ受信の終了時に実行されます。

名前
probe::socket.writev – **socket_writev** から送信したメッセージ

概要

```
socket.writev
```

値

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
メッセージサイズ (バイト単位)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージの送信者

説明
sock_writev 関数から、ソケットでのメッセージ送信の開始時に実行されます。

名前
probe::socket.writev.return – **socket_writev** から送信したメッセージの結論

概要

```
socket.writev.return
```

値

success
正常に送信されたか？(1 = yes、0 = no)

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
送信されるメッセージのサイズ (バイト単位) またはエラーコード (success = 0 の場合)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージ受信者。

説明
sock_writev 関数から、ソケットでのメッセージ送信の終了時に実行されます。

名前
probe::socket.readv – **sock_readv** からのメッセージの受信

概要

```
socket.readv
```

値

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
メッセージサイズ (バイト単位)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージの送信者

説明
sock_readv 関数からの、ソケットでのメッセージ受信の開始時に実行されます。

名前
probe::socket.readv.return – **sock_readv** からのメッセージの受信の結論

概要

■

```
socket.readv.return
```

値

success
正常に受信されたかどうか (1 = yes、0 = no)。(1 = yes、0 = no)

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前。

state
ソケット状態値

size
受信されるメッセージのサイズ (バイト単位) またはエラーコード (success = 0 の場合)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
メッセージ受信者。

説明
sock_readv 関数からの、ソケットでのメッセージ受信の終了時に実行されます。

名前
probe::socket.create – ソケットの作成。

概要

```
socket.create
```

値

protocol
プロトコル値

name
このプローブの名前

requester
ユーザープロセスまたはカーネルによる要求 (1 = kernel, 0 = user)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
リクエスター (リクエスター変数を参照)

説明
ソケットの作成開始時に実行されます。

名前
probe::socket.create.return – ソケットの作成から返します。

概要

```
socket.create.return
```

値

success
ソケットが正常に作成されたかどうか(1 = yes、0 = no)

protocol
プロトコル値

err
success == 0 の場合のエラーコード

name
このプローブの名前

requester
ユーザープロセスまたはカーネルによる要求 (1 = kernel, 0 = user)

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
リクエスター (ユーザープロセスまたはカーネル)

説明
ソケットの作成終了時に実行されます

名前
probe::socket.close – ソケットを閉じます

概要

```
socket.close
```

値

protocol
プロトコル値

flags
ソケットフラグ値

name
このプローブの名前

state
ソケット状態値

type
ソケットタイプの値。

family
プロトコルファミリー値

コンテキスト
リクエスター (ユーザープロセスまたはカーネル)

説明
ソケットのクローズ開始時に実行されます。

名前

probe::socket.close.return – ソケットのクローズから返します。

概要

```
socket.close.return
```

値

name

このプローブの名前

コンテキスト

リクエスター (ユーザープロセスまたはカーネル)

説明

ソケットのクローズ終了時に実行されます。

名前

function::sock_prot_num2str – プロトコル番号が指定される場合、文字列表現を返します。

概要

```
function sock_prot_num2str:string(proto:long)
```

引数

proto

プロトコル番号。

名前

function::sock_prot_str2num – プロトコル名 (文字列) が指定される場合、対応するプロトコル番号を返します。

概要

```
function sock_prot_str2num:long(proto:string)
```

引数

proto

プロトコル名。

名前
function::sock_fam_num2str – プロトコルファミリー番号が指定される場合、文字列表示を返します。

概要

```
function sock_fam_num2str:string(family:long)
```

引数

family
ファミリー番号。

名前
function::sock_fam_str2num – プロトコルファミリー名 (文字列) が指定される場合、対応する名前を返します。

概要

```
function sock_fam_str2num:long(family:string)
```

引数

family
ファミリー名。

説明
プロトコルファミリー番号。

名前
function::sock_state_num2str – ソケット状態番号が指定される場合、文字列表現を返します。

概要

```
function sock_state_num2str:string(state:long)
```

引数

state
状態番号。

名前

function::sock_state_str2num – ソケット状態文字列が指定される場合、対応する状態番号を返します。

概要

```
function sock_state_str2num:long(state:string)
```

引数

state

状態名。

第13章 カーネルプロセス TAPSET

この種類のプローブポイントは、プロセス関連のアクティビティーをプローブするために使用されます。以下のプローブポイントが含まれます。

名前
probe::kprocess.create – 新規プロセスが正常に作成されたときに必ず実行されます。

概要

```
kprocess.create
```

値

new_pid

新規に作成されたプロセスの PID。

コンテキスト
作成されたプロセスの親。

説明
フォーク (またはシステムコールのいずれか) または新しいカーネルスレッドの結果、新規プロセスが正常に作成されたときに必ず実行されます。

名前
probe::kprocess.start – 新規プロセスを開始します。

概要

```
kprocess.start
```

値
なし

コンテキスト
新規に作成されたプロセス。

説明
新規プロセスが実行を開始する直前に実行されます。

名前
probe::kprocess.exec – 新しいプログラムの実行の試行。

概要

```
kprocess.exec
```

値

filename

新しい実行可能ファイルのパス。

コンテキスト

exec の呼び出し元。

説明

プロセスが新規プログラムの実行を試みるたびに実行されます。

名前

probe::kprocess.exec_complete – 新しいプログラムの実行から返されます。

概要

```
kprocess.exec_complete
```

値

success

実行が成功したかどうかを示すブール値。

errno

実行の結果返されたエラー番号。

コンテキスト

成功した場合は、新しい実行可能ファイルのコンテキスト。失敗した場合は、呼び出し元のコンテキストに留まります。

説明

実行呼び出しの完了時に実行されます。

名前

probe::kprocess.exit – プロセスを終了します。

概要

```
kprocess.exit
```

値

code

code

プロセスの終了コード。

コンテキスト

終了するプロセス。

説明

プロセスが終了すると実行されます。この後で kprocess.release が常に実行されます。ただし、プロセスがゾンビ状態で待機している場合、kprocess.release の実行は遅延することがあります。

名前

probe::kprocess.release – リリースされるプロセス。

概要

```
kprocess.release
```

値

pid

リリースされるプロセスの PID。

task

リリースされるプロセスのタスクハンドル。

コンテキスト

親のコンテキスト (このプロセスの終了を通知したい場合。それ以外の場合は、プロセス自体のコンテキスト)。

説明

プロセスがカーネルからリリースされると実行されます。これは kprocess.exit に従うことがありますが、プロセスが異常状態で待機している場合に若干遅れる可能性があります。

第14章 シグナルTAPSET

この種類のプローブポイントは、シグナルアクティビティーをプローブするために使用されます。以下のプローブポイントが含まれます。

名前

probe::signal.send – プロセスに送信されるシグナル。

概要

```
signal.send
```

値

send2queue

シグナルが既存の sigqueue に送信されたかどうかを示します。

name

シグナルを送信するために使用される関数の名前。

task

シグナル受信者のタスクハンドル

sinfo

siginfo 構造のアドレス

si_code

シグナルタイプを示します。

sig_name

シグナルの文字列表現

sig

シグナルの番号

shared

シグナルがスレッドグループで共有されているかどうかを示します。

sig_pid

シグナルを受信するプロセスの PID

pid_name

シグナル受信者の名前

コンテキスト

シグナルの送信者。

名前
probe::signal.send.return – 完了したプロセスに送信されているシグナル

概要

```
signal.send.return
```

値

retstr

__group_send_sig_info、specific_send_sig_info、または send_sigqueue への戻り値

send2queue

送信されたシグナルが既存の sigqueue に送信されたかどうかを示します。

name

シグナルを送信するために使用される関数の名前。

shared

送信されたシグナルがスレッドグループで共有されているかどうかを示します。

コンテキスト
シグナルの送信者 (正しいか?)。

説明
__group_send_sig_info と specific_send_sig_info の戻り値は以下のとおりです。

0 -- シグナルがプロセスに正常に送信されました。

つまり、
(1) シグナルが受信側プロセスによって無視されました。(2) これは非 RT シグナルであり、システムのキューにはすでに 1 つのシグナルが格納されています。(3) シグナルが受信側プロセスの sigqueue に正常に追加されました。

-EAGAIN -- 受信側プロセスの sigqueue はオーバーフロー状態です。シグナルは RT であり、**kill** 以外の関数を使用しているユーザーによって送信されました。

send_group_sigqueue と send_sigqueue の戻り値は以下のとおりです。

0 -- シグナルは受信側プロセスの sigqueue に正常に追加されました。または、SI_TIMER エントリーがすでにキューに格納されています (この場合、オーバーランした数は単純に増分されます)。

1 -- シグナルが受信側プロセスによって無視されました。

-1 -- (send_sigqueue のみ) タスクは終了中とマークされ、* posix_timer_event をグループリーダーにリダイレクトすることが許可されます。

名前
probe::signal.checkperm – 送信されたシグナルについての確認が実行されます。

概要

```
signal.checkperm
```

値

name
プローブポイントの名前。

task
シグナル受信者のタスクハンドル

sinfo
siginfo 構造のアドレス

si_code
シグナルタイプを示します。

sig_name
シグナルの文字列表現

sig
シグナルの番号

pid_name
シグナルを受信するプロセスの名前

sig_pid
シグナルを受信するプロセスの PID

名前
probe::signal.checkperm.return – 送信されたシグナルの確認実行が完了しました。

概要

```
signal.checkperm.return
```

値

retstr
値を文字列として返します。

name

プローブポイントの名前。

名前

probe::signal.wakeup – シグナルによりウェイクするスリープ状態のプロセス

概要

```
signal.wakeup
```

値

resume

STOPPED または TRACED 状態のタスクをウェイクアップするかどうかを示します。

state_mask

ウェイクするタスク状態のマスクを示す文字列表現。使用できる値は TASK_INTERRUPTIBLE、TASK_STOPPED、TASK_TRACED、および TASK_INTERRUPTIBLE です。

pid_name

ウェイクするプロセスの名前。

sig_pid

ウェイクするプロセスの PID

名前

probe::signal.check_ignored – シグナルが無視されたことを確認します。

概要

```
signal.check_ignored
```

値

sig_name

シグナルの文字列表現

sig

シグナルの番号

pid_name

シグナルを受信するプロセスの名前

sig_pid
シグナルを受信するプロセスの PID

名前
probe::signal.check_ignored.return – シグナルが無視されたことの確認が完了しました。

概要

```
signal.check_ignored.return
```

値

retstr
値を文字列として返します。

name
プローブポイントの名前。

名前
probe::signal.force_segv – IGSEGV の送信を強制実行します

概要

```
signal.force_segv
```

値

name
プローブポイントの名前。

sig_name
シグナルの文字列表現

sig
シグナルの番号

pid_name
シグナルを受信するプロセスの名前

sig_pid
シグナルを受信するプロセスの PID

名前
probe::signal.force_segv.return – SIGSEGV の送信の強制実行が完了しました。

概要

```
signal.force_segv.return
```

値

retstr
値を文字列として返します。

name
プローブポイントの名前

名前
probe::signal.syskill – プロセスに kill シグナルを送信します。

概要

```
signal.syskill
```

値

name
プローブポイントの名前

sig_name
シグナルの文字列表現

sig
プロセスに送信される特定のシグナル

pid_name
シグナル受信者の名前

sig_pid
シグナルを受信するプロセスの PID

名前
probe::signal.syskill.return – kill シグナルの送信が完了しました。

概要

```
signal.syskill.return
```

値
なし

名前
probe::signal.sys_tkill – スレッドに kill シグナルを送信します。

概要

```
signal.sys_tkill
```

値

name
プローブポイントの名前

sig_name
シグナルの文字列表現

sig
プロセスに送信される特定のシグナル

pid_name
シグナル受信者の名前

sig_pid
kill シグナルを受信するプロセスの PID

説明
tkill 呼び出しは kill(2) と似ていますが、特定のスレッドグループ内のプロセスを対象にすることができます。このようなプロセスは、一意のスレッド ID (TID) でターゲットとなります。

名前
probe::signal.systkill.return – スレッドへの kill シグナルの送信が完了しました。

概要

```
signal.systkill.return
```

値

retstr
__group_send_sig_info に対する戻り値。

name
プローブポイントの名前

名前
probe::signal.sys_tgkill – スレッドグループに kill シグナルを送信します。

概要

```
signal.sys_tgkill
```

値

name
プローブポイントの名前

sig_name
シグナルの文字列表現

sig
プロセスに送信される特定の kill シグナル。

tgid
kill シグナルを受信するスレッドのスレッドグループ ID

pid_name
シグナル受信者の名前

sig_pid
kill シグナルを受信するスレッドの PID

説明
tgkill 呼び出しは tkill に似ています。ただし、呼び出し元は、シグナルを送信するスレッドのスレッドグループ ID を指定できます。これにより、TID の再利用を回避できます。

名前
probe::signal.sys_tgkill.return – スレッドグループへの kill シグナルの送信が完了しました。

概要

```
signal.sys_tgkill.return
```

値

retstr

retstr

__group_send_sig_info に対する戻り値。

name

プローブポイントの名前

名前

probe::signal.send_sig_queue – シグナルをプロセスのキューに格納します。

概要

```
signal.send_sig_queue
```

値

sigqueue_addr

シグナルキューのアドレス

name

プローブポイントの名前

sig_name

シグナルの文字列表現

sig

キューに格納されたシグナル

pid_name

シグナルがキューに格納されるプロセスの名前

sig_pid

シグナルがキューに格納されるプロセスの PID

名前

probe::signal.send_sig_queue.return – プロセスのキューへのシグナルの格納が完了しました。

概要

```
signal.send_sig_queue.return
```

値

retstr

値を文字列として返します。

name
プローブポイントの名前

名前
probe::signal.pending – 保留中シグナルの調査

概要

```
signal.pending
```

値

name
プローブポイントの名前

sigset_size
ユーザー空間シグナルセットのサイズ

sigset_add
ユーザー空間シグナルセットのアドレス (sigset_t)

説明
このプローブは、特定のスレッドへの配信を保留しているシグナルのセットを調べるために使用されます。これは通常、do_sigpending カーネル関数を実行する際に発生します。

名前
probe::signal.pending.return – 保留中のシグナルの調査が完了しました。

概要

```
signal.pending.return
```

値

retstr
値を文字列として返します。

name
プローブポイントの名前

名前
probe::signal.handle – 呼び出されるシグナルハンドラー。

概要

```
signal.handle
```

値

regs
kernel-mode スタック領域のアドレス。

sig_code
siginfo シグナルの si_code 値。

name
プローブポイントの名前

sig_mode
シグナルがユーザーモードシグナルであるか、またはカーネルモードシグナルであるかを示します。

sinfo
siginfo テーブルのアドレス。

sig_name
シグナルの文字列表現

oldset_addr
ブロックされたシグナルのビットマスクアレイのアドレス。

sig
シグナルハンドラーを呼び出したシグナル番号。

ka_addr
シグナルに関連付けられた k_sigaction テーブルのアドレス

名前
probe::signal.handle.return – シグナルハンドラーの呼び出しが完了しました。

概要

```
signal.handle.return
```

値

retstr
値を文字列として返します。

name
プローブポイントの名前

名前
probe::signal.do_action – シグナルアクションを調査または変更します。

概要

```
signal.do_action
```

値

sa_mask
シグナルの新しいマスク。

name
プローブポイントの名前

sig_name
シグナルの文字列表現

oldsigact_addr
シグナルに関連付けられた古い sigaction 構造のアドレス

sig
調査または変更するシグナル。

sa_handler
シグナルの新しいハンドラー。

sigact_addr
シグナルに関連付けられた新しい sigaction 構造のアドレス。

名前
probe::signal.do_action.return – シグナルアクションの調査または変更が完了しました。

概要

```
signal.do_action.return
```

値

retstr

値を文字列として返します。

name

プローブポイントの名前

名前

probe::signal.procmask – ブロックされたシグナルを調査または変更します。

概要

```
signal.procmask
```

値

how

ブロックされたシグナルを変更する方法を示します。値は SIG_BLOCK=0 (シグナルをブロックする場合)、SIG_UNBLOCK=1 (シグナルをブロック解除する場合)、および SIG_SETMASK=2 (シグナルマスクを設定する場合) です。

name

プローブポイントの名前

oldsigset_addr

シグナルセット (sigset_t) の古いアドレス。

sigset

sigset_t に設定する実際の値 (正しいか?)。

sigset_addr

実装するシグナルセット (sigset_t) のアドレス。

名前

probe::signal.procmask.return – ブロックされたシグナルの調査または変更が完了しました。

概要

```
signal.procmask.return
```

値

retstr

値を文字列として返します。

name

プローブポイントの名前

名前

probe::signal.flush – タスクのすべての保留中シグナルを破棄します。

概要

```
signal.flush
```

値

name

プローブポイントの名前

task

破棄を実行するプロセスのタスクハンドラー

pid_name

破棄を実行するタスクに関連付けられたプロセスの名前

sig_pid

破棄を実行するタスクに関連付けられたプロセスの PID

第15章 DIRECTORY-ENTRY (DENTRY) TAPSET

この種類の関数は、ファイルまたは完全パス名にカーネル VFS ディレクトリーのエントリーポインターをマップするために使用されます。

名前

function::d_name – dirent 名を取得します。

概要

```
function d_name:string(dentry:long)
```

引数

dentry

dentry へのポインター。

説明

dirent 名 (パスベース名) を返します。

名前

function::reverse_path_walk – 完全 dirent パスを取得します。

概要

```
function reverse_path_walk:string(dentry:long)
```

引数

dentry

dentry へのポインター。

説明

パス名を返します (マウントポイントへの部分パス)

名前

function::task_dentry_path – 完全 dentry パスを取得します。

概要

```
function task_dentry_path:string(task:long,dentry:long,vfsmnt:long)
```

引数

task

task_struct ポインター

dentry

direntry ポインター。

vfsmnt

vfsmnt ポインター。

説明

kernel d_path 関数などの完全 dirent 名を返します (root への完全パス)。

名前

function::d_path – 完全 nameidata パスを取得します。

概要

```
function d_path:string(nd:long)
```

引数

nd

nameidata へのポインター。

説明

kernel d_path 関数などの完全 dirent 名を返します (root への完全パス)。

第16章 ロギングTAPSET

この種類の関数は、単純なメッセージ文字列を各種の宛先に送信するために使用されます。

名前
function::log – 共通トレースバッファーに行を送信します。

概要

```
function log(msg:string)
```

引数

msg

フォーマットされたメッセージ文字列。

一般的な構文
log(msg:string)

説明
この関数はデータをログに記録します。ログはメッセージをすぐに staprun と、一括トランスポート (relayfs) (使用している場合) に送信します。最後の文字が改行でない場合は、追加されます。この関数は printf ほど効率的ではありません。緊急メッセージにのみ使用してください。

名前
function::warn – 警告ストリームに行を送信します。

概要

```
function warn(msg:string)
```

引数

msg

フォーマットされたメッセージ文字列。

一般的な構文
warn(msg:string)

説明
この関数は警告メッセージをすぐに staprun に送信します。また、一括トランスポート (relayfs) (使用されている場合) にわたり送信されます。最後の文字が改行でない場合は、その文字が追加されます。

名前
function::exit – プローブスクリプトのシャットダウンを開始します。

概要

```
function exit()
```

引数
なし

一般的な構文
exit

説明
これにより、スクリプトのシャットダウンを開始する要求のみをキューに入れます。新規プローブは実行されません (「end」 プローブを除く)。現在実行中のプローブはすべて、その作業を完了する可能性があります。

名前
function::error – エラーメッセージを送信します。

概要

```
function error(msg:string)
```

引数

msg

フォーマットされたメッセージ文字列。

説明
暗黙的な行末が追加されます。staprun は文字列 「ERROR:」 の前に付けられます。エラーメッセージを送信すると、現在実行中のプローブが中止します。MAXERRORS パラメーターによっては、**exit** がトリガーされる可能性があります。

名前
function::ftrace – ftrace リングバッファーにメッセージを送信します。

概要

```
function ftrace(msg:string)
```

引数

msg

フォーマットされたメッセージ文字列。

説明

ftrace リングバッファーが設定され、利用可能な場合は、メッセージについては /debugfs/tracing/trace を参照してください。そうでない場合は、メッセージは警告なしで破棄される可能性があります。暗黙的な行末が追加されます。

第17章 ランダム関数 TAPSET

以下の関数は乱数の生成を行います。

名前

function::randint – [0,n) の範囲のランダム数を返します。

概要

```
function randint:long(n:long)
```

引数

n

範囲の上限を超える数字 (2**20 を超えない)。

第18章 文字列およびデータ取得関数TAPSET

アドレスに基づいてカーネルまたはユーザー空間プログラムから文字列およびその他のプリミティブタイプを取得する関数。すべての文字列は MAXSTRINGLEN で指定される最大長です。

名前
function::kernel_string – カーネルメモリーから文字列を取得します。

概要

```
function kernel_string:string(addr:long)
```

引数

addr

文字列の取得元のカーネルアドレス。

一般的な構文
kernel_string:string(addr:long)

説明
この関数は、指定のカーネルメモリーアドレスから NULL 終端 C 文字列を返します。文字列のコピー障害のエラーを報告します。

名前
function::kernel_string2 – 代替エラー文字列と共にカーネルメモリーから文字列を取得します。

概要

```
function kernel_string2:string(addr:long,err_msg:string)
```

引数

addr

文字列の取得元のカーネルアドレス。

err_msg

データが利用できない場合に返すエラーメッセージ。

一般的な構文
kernel_string2:string(addr:long, err_msg:string)

説明
この関数は、指定のカーネルメモリーアドレスから NULL 終端 C 文字列を返します。文字列のコピー障害の所定のエラーメッセージを報告します。

名前
function::kernel_string_n – カーネルメモリーから所定の長さの文字列を取得します。

概要

```
function kernel_string_n:string(addr:long,n:long)
```

引数

addr

文字列の取得元のカーネルアドレス。

n

文字列の最大長 (null で終了していない場合)。

一般的な構文
kernel_string_n:string(addr:long, n:long)

説明
指定のカーネルメモリーアドレスから最大長の C 文字列を返します。文字列のコピー障害のエラーを報告します。

名前
function::kernel_long – カーネルメモリーに保存された long 値を取得します。

概要

```
function kernel_long:long(addr:long)
```

引数

addr

long 値の取得元のカーネルアドレス。

一般的な構文
kernel_long:long(addr:long)

説明
指定のカーネルメモリーアドレスから long 値を返します。指定アドレスからの読み取りに失敗する場合にエラーを報告します。

名前

function::kernel_int – カーネルメモリーに保存される int 値を取得します。

概要

```
function kernel_int:long(addr:long)
```

引数

addr

int の取得元のカーネルアドレスです。

説明

指定のカーネルメモリーアドレスから int 値を返します。指定アドレスからの読み取りに失敗する場合にエラーを報告します。

名前

function::kernel_short – カーネルメモリーに保存される short 値を取得します。

概要

```
function kernel_short:long(addr:long)
```

引数

addr

short 値の取得元のカーネルアドレス。

一般的な構文

kernel_short:long(addr:long)

説明

指定のカーネルメモリーアドレスから short 値を返します。指定アドレスからの読み取りに失敗する場合にエラーを報告します。

名前

function::kernel_char – カーネルメモリーに保存された char 値を取得します。

概要

```
function kernel_char:long(addr:long)
```

引数

addr

char 値の取得元のカーネルアドレス。

一般的な構文
kernel_char:long(addr:long)

説明
指定のカーネルメモリーアドレスから char 値を返します。指定アドレスからの読み取りに失敗する場合にエラーを報告します。

名前
function::kernel_pointer – カーネルメモリーに保存されるポインター値を取得します。

概要

```
function kernel_pointer:long(addr:long)
```

引数

addr

ポインターの取得元のカーネルアドレス。

一般的な構文
kernel_pointer:long(addr:long)

説明
指定のカーネルメモリーアドレスからポインター値を返します。指定アドレスからの読み取りに失敗する場合にエラーを報告します。

名前
function::user_string – ユーザー空間から文字列を取得します。

概要

```
function user_string:string(addr:long)
```

引数

addr

文字列の取得元のユーザー空間アドレス。

一般的な構文
user_string:string(addr:long)

説明

指定のユーザー空間メモリーアドレスから NULL 終端 C 文字列を返します。ユーザー空間データにアクセスできない稀なケースでは 「<unknown>」 を報告します。

名前
function::user_string2 – 代替エラー文字列と共にユーザー空間から文字列を取得します。

概要

```
function user_string2:string(addr:long,err_msg:string)
```

引数

addr

文字列の取得元のユーザー空間アドレス。

err_msg

データが利用できない場合に返すエラーメッセージ。

一般的な構文
user_string2:string(addr:long, err_msg:string)

説明
指定のユーザー空間メモリーアドレスから NULL 終端 C 文字列を返します。ユーザー空間データにアクセスできない稀なケースで所定のエラーメッセージを報告します。

名前
function::user_string_warn – ユーザー空間から文字列を取得します。

概要

```
function user_string_warn:string(addr:long)
```

引数

addr

文字列の取得元のユーザー空間アドレス。

一般的な構文
user_string_warn:string(addr:long)

説明
指定のユーザー空間メモリーアドレスから NULL 終端 C 文字列を返します。ユーザー空間データにアクセスできない場合に、まれなケースで 「<unknown>」 を報告し、障害に関して警告します (ただし、中止しません)。

名前
function::user_string_quoted – ユーザー空間から文字列を取得し、引用符で囲みます。

概要

```
function user_string_quoted:string(addr:long)
```

引数

addr
文字列の取得元のユーザー空間アドレス。

一般的な構文
user_string_quoted:string(addr:long)

説明
指定のユーザー空間メモリーアドレスから NULL 終端 C 文字列を返します。出力不可能な ASCII 文字は、返される文字列の対応するエスケープシーケンスに置き換えられます。アドレスゼロの 「NULL」 を報告します。指定のアドレスでユーザー空間データにアクセスできない稀なケースで 「<unknown>」 を返します。

名前
function::user_string_n – ユーザー空間から所定の長さの文字列を取得します。

概要

```
function user_string_n:string(addr:long,n:long)
```

引数

addr
文字列の取得元のユーザー空間アドレス。

n
文字列の最大長 (null で終了していない場合)。

一般的な構文
user_string_n:string(addr:long, n:long)

説明
指定のユーザー空間アドレスから最大長の C 文字列を返します。指定のアドレスでユーザー空間データにアクセスできない稀なケースで 「<unknown>」 を返します。

名前

function::user_string_n2 – ユーザー空間から所定の長さの文字列を取得します。

概要

```
function user_string_n2:string(addr:long,n:long,err_msg:string)
```

引数

addr

文字列の取得元のユーザー空間アドレス。

n

文字列の最大長 (null で終了していない場合)。

err_msg

データが利用できない場合に返すエラーメッセージ。

一般的な構文

user_string_n2:string(addr:long, n:long, err_msg:string)

説明

指定のユーザー空間アドレスから最大長の C 文字列を返します。指定のアドレスでユーザー空間データにアクセスできない稀なケースで指定のエラーメッセージ文字列を返します。

名前

function::user_string_n_warn – ユーザー空間から文字列を取得します。

概要

```
function user_string_n_warn:string(addr:long,n:long)
```

引数

addr

文字列の取得元のユーザー空間アドレス。

n

文字列の最大長 (null で終了していない場合)。

一般的な構文

user_string_n_warn:string(addr:long, n:long)

説明

指定のユーザー空間メモリーアドレスから C 文字列の n 文字までを返します。ユーザー空間データにアクセスできない場合に、まれなケースで 「<unknown>」 を報告し、障害に関して警告します (ただし、中止しません)。

名前
function::user_string_n_quoted – ユーザー空間から文字列を取得し、引用符で囲みます。

概要

```
function user_string_n_quoted:string(addr:long,n:long)
```

引数

addr

文字列の取得元のユーザー空間アドレス。

n

文字列の最大長 (null で終了していない場合)。

一般的な構文
user_string_n_quoted:string(addr:long, n:long)

説明
指定のユーザー空間メモリーアドレスから n 文字の 終端 C 文字列を返します。出力不可能な ASCII 文字は、返される文字列の対応するエスケープシーケンスに置き換えられます。アドレスゼロの「NULL」 を報告します。指定のアドレスでユーザー空間データにアクセスできない稀なケースで「<unknown>」 を返します。

名前
function::user_short – ユーザー空間に保存された short 値を取得します。

概要

```
function user_short:long(addr:long)
```

引数

addr

short 値の取得元のユーザー空間アドレスです。

一般的な構文
user_short:long(addr:long)

説明
指定のユーザー空間アドレスから short 値を返します。ユーザー空間データにアクセスできない場合はゼロを返します。

名前
function::user_short_warn – ユーザー空間に保存された short 値を取得します。

概要

```
function user_short_warn:long(addr:long)
```

引数

addr

short 値の取得元のユーザー空間アドレスです。

一般的な構文
user_short_warn:long(addr:long)

説明
指定のユーザー空間アドレスから short 値を返します。ユーザー空間データにアクセスできない場合はゼロを返し、障害に関して警告します (ただし、中止しません)。

名前
function::user_int – ユーザー空間に保存された int 値を取得します。

概要

```
function user_int:long(addr:long)
```

引数

addr

int 値の取得元のユーザー空間アドレス。

一般的な構文
user_int:long(addr:long)

説明
指定のユーザー空間アドレスから int 値を返します。ユーザー空間データにアクセスできない場合はゼロを返します。

名前
function::user_int_warn – ユーザー空間に保存された int 値を取得します。

概要

```
function user_int_warn:long(addr:long)
```

引数

addr

int 値の取得元のユーザー空間アドレス。

一般的な構文

user_ing_warn:long(addr:long)

説明

指定のユーザー空間アドレスから int 値を返します。ユーザー空間データにアクセスできない場合はゼロを返し、障害に関して警告します (ただし、中止しません)。

名前

function::user_long – ユーザー空間に保存された long 値を取得します。

概要

```
function user_long:long(addr:long)
```

引数

addr

long 値の取得元のユーザー空間アドレス。

一般的な構文

user_long:long(addr:long)

説明

指定のユーザー空間アドレスから long 値を返します。ユーザー空間データにアクセスできない場合はゼロを返します。longのサイズは、現在のユーザー空間タスクのアーキテクチャーによって異なることに注意してください (64/32 ビット互換タスクの両方をサポートするアーキテクチャーの場合)。

名前

function::user_long_warn – ユーザー空間に保存された long 値を取得します。

概要

```
function user_long_warn:long(addr:long)
```

引数

addr

long 値の取得元のユーザー空間アドレス。

一般的な構文
user_long_warn:long(addr:long)

説明
指定のユーザー空間アドレスから long 値を返します。ユーザー空間データにアクセスできない場合はゼロを返し、障害に関して警告します (ただし、中止しません)。longのサイズは、現在のユーザー空間タスクのアーキテクチャーによって異なることに注意してください (64/32 ビット互換タスクの両方をサポートするアーキテクチャーの場合)。

名前
function::user_char – ユーザー空間に保存された char 値を取得します。

概要

```
function user_char:long(addr:long)
```

引数

addr

char の取得元のユーザー空間アドレス。

一般的な構文
user_char:long(addr:long)

説明
指定のユーザー空間アドレスから char 値を返します。ユーザー空間データにアクセスできない場合はゼロを返します。

名前
function::user_char_warn – ユーザー空間に保存された char 値を取得します。

概要

```
function user_char_warn:long(addr:long)
```

引数

addr

char の取得元のユーザー空間アドレス。

一般的な構文
user_char_warn:long(addr:long)

説明

指定のユーザー空間アドレスから char 値を返します。ユーザー空間データにアクセスできない場合はゼロを返し、障害に関して警告します (ただし、中止しません)。

第19章 標準的な文字列関数のコレクション

長さ、サブ文字列の取得、個別の文字の取得、文字列の検索、エスケープ、トークン化および文字列の long への変換を実行する関数です。

名前
function::strlen – 文字列の長さを返します。

概要

```
function strlen:long(s:string)
```

引数

s

文字列。

一般的な構文
strlen: long (str:string)

説明
この関数は、ゼロから MAXSTRINGLEN までに設定できる文字列の長さを返します。

名前
function::substr – サブ文字列を返します。

概要

```
function substr:string(str:string,start:long,length:long)
```

引数

str

サブ文字列の取得元の文字列

start

開始位置。0 = 文字列の開始点。

length

返す文字列の長さ。

一般的な構文
substr:string (str:string, start:long, stop:long)

説明

指定の開始位置から始まり、指定の指定の停止位置で終了する、指定の長さまでのサブ文字列を返します。

名前
function::stringat – 文字列の所定位置の文字を返します。

概要

```
function stringat:long(str:string,pos:long)
```

引数

str

文字の取得元の文字列。

pos

文字の取得元の位置。0 = 文字列の開始点。

一般的な構文
stringat:long(srt:string, pos:long)

説明
この関数は、文字列の指定の位置にある文字を返します。または、文字列に多くの文字がない場合はゼロを返します。

名前
function::isinstr – 文字列が別の文字列のサブ文字列かどうかを返します。

概要

```
function isinstr:long(s1:string,s2:string)
```

引数

s1

検索する文字列。

s2

検索するサブ文字列。

一般的な構文
isinstr:long(s1:string, s2:string)

説明

この関数は、文字列 s1 に s2 が含まれる場合に 1 を返します。そうでない場合はゼロを返します。

名前
function::text_str – 文字列の出力できない文字をエスケープします。

概要

```
function text_str:string(input:string)
```

引数

input
エスケープする文字列。

一般的な構文
text_str:string (input:string)

説明
この関数は文字列引数を受け入れ、出力不可能なすべての ASCII 文字は、返される文字列の対応するエスケープシーケンスに置き換えられます。

名前
function::text_strn – 文字列の出力できない文字をエスケープします。

概要

```
function text_strn:string(input:string,len:long,quoted:long)
```

引数

input
エスケープする文字列。

len
返す文字列の最大長。0 は MAXSTRINGLEN を意味します。

quoted
文字列を二重引用符で囲みます。入力文字列が切り捨てられる場合は、2 つ目の引用符の後に 「...」 が使用されます。

一般的な構文
text_strn:string(input:string, len:long, quoted:long)

説明

この関数は指定された長さの文字列を受け入れ、出力不可能なすべての ASCII 文字は、返される文字列の対応するエスケープシーケンスに置き換えられます。

名前
function::tokenize – 文字列の次の空でないトークンを返します。

概要

```
function tokenize:string(input:string,delim:string)
```

引数

input

トークン化する文字列。NULL の場合は、**tokenize** を行う直前の呼び出しで渡された文字列の空でないトークンを返します。

delim

トークン区切り文字。トークンを区切る文字セット。

一般的な構文
tokenize:string(input:string, delim:string)

説明
この関数は、トークンが delim 文字列の文字で区切られる指定の入力文字列の次の空でないトークンを返します。入力文字列が NULL 以外の場合、最初のトークンを返します。入力文字列が NULL の場合は、トークン化する直前のへの呼び出しで渡された文字列の次のトークンを返します。区切り文字が見つからない場合は、残りの入力文字列全体が返されます。利用可能なトークンがない場合には NULL を返します。

名前
function::str_replace – str_replace は、サブ文字列のすべてのインスタンスを別のものに置き換えます。

概要

```
function str_replace:string(prnt_str:string,srch_str:string,rplc_str:string)
```

引数

prnt_str

検索し、置換する文字列。

srch_str

prnt_str 文字列で検索するために使用されるサブ文字列。

rplc_str

srch_str を置き換えるために使用されるサブ文字列。

一般的な構文

str_replace:string(prnt_str:string, srch_str:string, rplc_str:string)

説明

この関数は、サブ文字列が置換された所定の文字列を返します。

名前

function::strtol – strtol - 文字列を long に変換します。

概要

```
function strtol:long(str:string,base:long)
```

引数

str

変換する文字列。

base

使用するベース。

一般的な構文

strtol:long(str:string, base:long)

説明

この関数は、数字の文字列表現を整数に変換します。ベースパラメーターは文字列に想定する数値のベースを示します (例: 16 進数の場合は 16、8 進数の場合は 8、バイナリーは 2)。

名前

function::isdigit – 数字をチェックします。

概要

```
function isdigit:long(str:string)
```

引数

str

チェックする文字列。

一般的な構文

isdigit:long(str:string)

説明

文字列の最初の文字として数字 (0 から 9) があるかどうかを確認します。true の場合はゼロ以外の値を返し、false の場合はゼロを返します。

第20章 ANSI 制御文字をログで使用するためのユーティリティー関数

ansi 制御文字を使用してロギングするユーティリティー関数。これにより、カーソル位置と文字色出力、ログメッセージの属性を変更できます。

名前
function::ansi_clear_screen – カーソルを左上に移動し、画面をクリアします。

概要

```
function ansi_clear_screen()
```

引数
なし

一般的な構文
ansi_clear_screen

説明
カーソルを左上に移動するために ansi コードを送信し、カーソル位置から終了位置までの画面をクリアするために ansi コードを送信します。

名前
function::ansi_set_color – ansi Select Graphic Rendition モードを設定します。

概要

```
function ansi_set_color(fg:long)
```

引数

fg
設定する前景色。

一般的な構文
ansi_set_color(fh:long)

説明
指定のフォグラウンドカラー用に Select Graphic Rendition モードの ansi コードを送信します。黒(30)、青(34)、緑(32)、シアン(36)、赤(31)、紫(35)、緑(33)、緑(37)。

名前
function::ansi_set_color2 – ansi Select Graphic Rendition モードを設定します。

概要

function ansi_set_color2(fg:long,bg:long)

引数

fg

設定する前景色。

bg

設定する背景色。

一般的な構文

ansi_set_color2(fg:long, bg:long)

説明

前景色 (黒 (30)、青 (34)、緑 (32)、シアン (36)、赤 (31)、紫 (35)、茶 (33)、灰 (37)) および背景色 (黒 (40)、赤 (41)、緑 (42)、黄 (43)、青 (44)、紫 (45)、シアン (46)、白 (47)) を指定する Select Graphic Rendition モードの ANSI コードを送信します。

名前

function::ansi_set_color3 – ansi Select Graphic Rendition モードを設定します。

概要

function ansi_set_color3(fg:long,bg:long,attr:long)

引数

fg

設定する前景色。

bg

設定する背景色。

attr

設定する色属性。

一般的な構文

ansi_set_color3(fg:long, bg:long, attr:long)

説明

前景色 (黒 (30)、青 (34)、緑 (32)、シアン (36)、赤 (31)、紫 (35)、茶 (33)、灰 (37))、および背景色 (黒 (40)、赤 (41)、緑 (42)、黄 (43)、青 (44)、紫 (45)、シアン (46)、白 (47))、およびすべての属性を解除 (0)、太字 (1)、下線 (4)、点滅 (5)、高速点滅 (6)、 反転 (7)などの色属性を指定する Select Graphic Rendition モードの ANSI コードを送信します。

名前
function::ansi_reset_color – Select Graphic Rendition モードをリセットします。

概要

```
function ansi_reset_color()
```

引数
なし

一般的な構文
ansi_reset_color

説明
前景色、背景色および色属性をデフォルト値にリセットするために ansi コードを送信します。

名前
function::ansi_new_line – カーソルを新しい行に移動します。

概要

```
function ansi_new_line()
```

引数
なし

一般的な構文
ansi_new_line

説明
改行の ANSI コードを送信します。

名前
function::ansi_cursor_move – カーソルを新規の座標に移動します。

概要

```
function ansi_cursor_move(x:long,y:long)
```

引数

x
カーソルの移動先の行。

y
カーソルの移動先の列。

一般的な構文
ansi_curos_move(x:long, y:long)

説明
カーソルを x 行および y 列に配置するために ansi コードを送信します。座標は 1 から始まります。(1,1) は左上隅です。

名前
function::ansi_cursor_hide – カーソルを非表示にします。

概要

```
function ansi_cursor_hide()
```

引数
なし

一般的な構文
ansi_cusor_hide

説明
カーソルを非表示にするために ansi コードを送信します。

名前
function::ansi_cursor_save – カーソル位置を保存します。

概要

```
function ansi_cursor_save()
```

引数
なし

一般的な構文
ansi_cursor_save

説明
現在のカーソル位置を保存するために ansi コードを送信します。

名前

function::ansi_cursor_restore – 以前に保存されたカーソル位置を復元します。

概要

```
function ansi_cursor_restore()
```

引数
なし

一般的な構文
ansi_cursor_restore

説明
ansi_cursor_save で以前に保存した現在のカーソル位置を復元するために ansi コードを送信します。

名前
function::ansi_cursor_show – カーソルを表示します。

概要

```
function ansi_cursor_show()
```

引数
なし

一般的な構文
ansi_cursor_show

説明
カーソルを表示するために ansi コードを送信します。