

Systemtap 1.0

SystemTap

Beginners Guide

Introduction to SystemTap (for Fedora 10)



Don Domingo

William Cohen

Systemtap 1.0 SystemTap Beginners Guide

Introduction to SystemTap (for Fedora 10)

Edition 2.0

Author

Don Domingo

ddomingo@redhat.com

Author

William Cohen

wcohen@redhat.com

Copyright © 2009 Red Hat, Inc

This documentation 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.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

This guide provides basic instructions on how to use SystemTap to monitor different subsystems of Fedora 10 in finer detail. The *SystemTap Beginners Guide* is recommended for users who have taken [RHCT](https://www.redhat.com/courses/rh133_red_hat_linux_system_administration_and_rhct_exam/)¹ or have a similar level of expertise in Fedora 10.

¹ https://www.redhat.com/courses/rh133_red_hat_linux_system_administration_and_rhct_exam/

Preface	v
1. Document Conventions	v
1.1. Typographic Conventions	v
1.2. Pull-quote Conventions	vi
1.3. Notes and Warnings	vii
2. We Need Feedback!	vii
1. Introduction	1
1.1. Documentation Goals	1
1.2. SystemTap Capabilities	1
2. Using SystemTap	3
2.1. Installation and Setup	3
2.1.1. Installing SystemTap	3
2.1.2. Installing Required Kernel Information RPMs	3
2.1.3. Initial Testing	5
2.2. Generating Instrumentation for Other Computers	5
2.3. Running SystemTap Scripts	7
3. Understanding How SystemTap Works	11
3.1. Architecture	11
3.2. SystemTap Scripts	11
3.2.1. Event	13
3.2.2. Systemtap Handler/Body	15
3.3. Basic SystemTap Handler Constructs	18
3.3.1. Variables	18
3.3.2. Conditional Statements	19
3.3.3. Command-Line Arguments	21
3.4. Associative Arrays	21
3.5. Array Operations in SystemTap	22
3.5.1. Assigning an Associated Value	22
3.5.2. Reading Values From Arrays	22
3.5.3. Incrementing Associated Values	23
3.5.4. Processing Multiple Elements in an Array	23
3.5.5. Clearing/Deleting Arrays and Array Elements	24
3.5.6. Using Arrays in Conditional Statements	25
3.5.7. Computing for Statistical Aggregates	26
3.6. Tapsets	28
4. Useful SystemTap Scripts	29
4.1. Network	29
4.1.1. Network Profiling	29
4.1.2. Tracing Functions Called in Network Socket Code	31
4.1.3. Monitoring Incoming TCP Connections	32
4.1.4. Monitoring TCP Packets	32
4.1.5. Monitoring Network Packets Drops in Kernel	34
4.2. Disk	35
4.2.1. Summarizing Disk Read/Write Traffic	35
4.2.2. Tracking I/O Time For Each File Read or Write	37
4.2.3. Track Cumulative IO	39
4.2.4. I/O Monitoring (By Device)	41
4.2.5. Monitoring Reads and Writes to a File	42
4.2.6. Monitoring Changes to File Attributes	43

4.2.7. Periodically Print I/O Block Time	43
4.3. Profiling	44
4.3.1. Counting Function Calls Made	45
4.3.2. Call Graph Tracing	46
4.3.3. Determining Time Spent in Kernel and User Space	47
4.3.4. Monitoring Polling Applications	49
4.3.5. Tracking Most Frequently Used System Calls	51
4.3.6. Tracking System Call Volume Per Process	53
4.4. Identifying Contended User-Space Locks	55
5. Understanding SystemTap Errors	57
5.1. Parse and Semantic Errors	57
5.2. Run Time Errors and Warnings	59
6. References	61
A. Revision History	63
Index	65

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the [Liberation Fonts](https://fedorahosted.org/liberation-fonts/)¹ set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

Mono-spaced Bold

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

To see the contents of the file **my_next_bestselling_novel** in your current working directory, enter the **cat my_next_bestselling_novel** command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

Press **Enter** to execute the command.

Press **Ctrl+Alt+F1** to switch to the first virtual terminal. Press **Ctrl+Alt+F7** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **mono-spaced bold**. For example:

File-related classes include **filesystem** for file systems, **file** for files, and **dir** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

¹ <https://fedorahosted.org/liberation-fonts/>

Close to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — *username*, *domain.name*, *file-system*, *package*, *version* and *release*. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

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

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;
```

```
import javax.naming.InitialContext;

public class ExClient
{
    public static void main(String args[])
        throws Exception
    {
        InitialContext iniCtx = new InitialContext();
        Object          ref    = iniCtx.lookup("EchoBean");
        EchoHome        home   = (EchoHome) ref;
        Echo            echo    = home.create();

        System.out.println("Created Echo");

        System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
    }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.



Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' won't cause data loss but may cause irritation and frustration.



Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: <http://bugzilla.redhat.com/bugzilla/> against the product **Red_Hat_Enterprise_Linux 5**.

When submitting a bug report, be sure to mention the manual's identifier:

SystemTap_Beginners_Guide

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Introduction

SystemTap is a tracing and probing tool that allows users to study and monitor the activities of the operating system (particularly, the kernel) in fine detail. It provides information similar to the output of tools like **netstat**, **ps**, **top**, and **iostat**; however, SystemTap is designed to provide more filtering and analysis options for collected information.

For system administrators, SystemTap can be used as a performance monitoring tool for Fedora 10. It is most useful when other similar tools cannot precisely pinpoint a bottleneck in the system, requiring a deep analysis of kernel activity. In the same manner, application developers can also use SystemTap to monitor, in finer detail, how their application behaves within the Linux system.

1.1. Documentation Goals

SystemTap provides the infrastructure to monitor the running Linux kernel for detailed analysis. This can assist administrators and developers in identifying the underlying cause of a bug or performance problem.

Without SystemTap, monitoring the activity of a running kernel would require a tedious instrument, recompile, install, and reboot sequence. SystemTap is designed to eliminate this, allowing users to gather the same information by simply running user-written SystemTap scripts.

However, SystemTap was initially designed for users with intermediate to advanced knowledge of the kernel. This makes SystemTap less useful to administrators or developers with limited knowledge of and experience with the Linux kernel. Moreover, much of the existing SystemTap documentation is similarly aimed at knowledgeable and experienced users. This makes learning the tool similarly difficult.

To lower these barriers the SystemTap Beginners Guide was written with the following goals:

- To introduce users to SystemTap, familiarize them with its architecture, and provide setup instructions for all kernel types.
- To provide pre-written SystemTap scripts for monitoring detailed activity in different components of the system, along with instructions on how to run them and analyze their output.

1.2. SystemTap Capabilities

SystemTap was originally developed to provide functionality for Fedora 10 similar to previous Linux probing tools such as **dprobes** and the Linux Trace Toolkit. SystemTap aims to supplement the existing suite of Linux monitoring tools by providing users with the infrastructure to track kernel activity. In addition, SystemTap combines this capability with two things:

- **Flexibility:** SystemTap's framework allows users to develop simple scripts for investigating and monitoring a wide variety of kernel functions, system calls, and other events that occur in kernel-space. With this, SystemTap is not so much a *tool* as it is a system that allows you to develop your own kernel-specific forensic and monitoring tools.
- **Ease-Of-Use:** as mentioned earlier, SystemTap allows users to probe kernel-space events without having to resort to instrument, recompile, install, and reboot the kernel.

Most of the SystemTap scripts enumerated in [Chapter 4, Useful SystemTap Scripts](#) demonstrate system forensics and monitoring capabilities not natively available with other similar tools (such

as **top**, **oprofile**, or **ps**). These scripts are provided to give readers extensive examples of the application of SystemTap, which in turn will educate them further on the capabilities they can employ when writing their own SystemTap scripts.

Limitations

The current iteration of SystemTap allows for a multitude of options when probing kernel-space events. However, SystemTap's ability to probe user-space events is quite limited. At present, the developmental efforts of the SystemTap community are geared towards improving SystemTap's user-space probing capabilities.

Using SystemTap

This chapter instructs users how to install SystemTap, and provides an introduction on how to run SystemTap scripts.

2.1. Installation and Setup

To deploy SystemTap, you need to install the SystemTap packages along with the corresponding set of **-devel**, **-debuginfo** and **-debuginfo-common** packages for your kernel. If your system has multiple kernels installed, and you wish to use SystemTap on more than one kernel, you will need to install the **-devel** and **-debuginfo** packages for *each* of those kernel versions.

These procedures will be discussed in detail in the following sections.



Important

Many users confuse **-debuginfo** with **-debug**. Remember that the deployment of SystemTap requires the installation of the **-debuginfo** package of the kernel, not the **-debug** version of the kernel.

2.1.1. Installing SystemTap

To deploy Systemtap, you will need to install the following RPMs:

- **systemtap**
- **systemtap-runtime**

Assuming that **yum** is installed in the system, these two rpms can be installed with **yum install systemtap systemtap-runtime**. Note that before you can use SystemTap, you will still need to install the required kernel information RPMs.

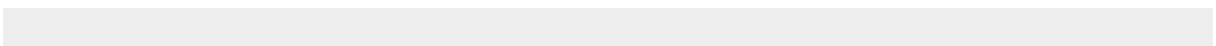
2.1.2. Installing Required Kernel Information RPMs

SystemTap needs information about the kernel in order to place instrumentation in it (i.e. probe it). This information also allows SystemTap to generate the code for the instrumentation. This information is contained in the matching **-devel**, **-debuginfo**, and **-debuginfo-common** packages for your kernel. The necessary **-devel** and **-debuginfo** packages for the ordinary "vanilla" kernel are as follows:

- **kernel-debuginfo**
- **kernel-debuginfo-common**
- **kernel-devel**

Likewise, the necessary packages for the PAE kernel would be **kernel-PAE-debuginfo**, **kernel-PAE-debuginfo-common**, and **kernel-PAE-devel**.

To determine what kernel your system is currently using, use:



```
uname -r
```

For example, if you wish to use SystemTap on kernel version **2.6.18-53.el5** on an i686 machine, then you would need to download and install the following RPMs:

- **kernel-debuginfo-2.6.18-53.1.13.el5.i686.rpm**
- **kernel-debuginfo-common-2.6.18-53.1.13.el5.i686.rpm**
- **kernel-devel-2.6.18-53.1.13.el5.i686.rpm**



Important

The version, variant, and architecture of the **-devel**, **-debuginfo** and **-debuginfo-common** packages must match the kernel you wish to probe with SystemTap *exactly*.

The easiest way to install the required kernel information packages is through **yum install debuginfo-install**. **debuginfo-install** is included with later versions of the **yum-utils** package (for example, version 1.1.10), and also requires an appropriate **yum** repository from which to download and install **-debuginfo/-debuginfo-common** packages.

Most of the required kernel packages can be found at <http://download.fedoraproject.org/pub/fedora/linux/releases/>. Configure **yum** accordingly by adding a new "debug" **yum** repository file under **/etc/yum.repos.d** containing the following lines:

```
[fedora-debuginfo]
name=Fedora $releasever - $basearch - Debug
failovermethod=priority
baseurl=http://download.fedoraproject.org/pub/fedora/linux/releases/$releasever/Everything/
$basearch/debug/
enabled=1
```

After configuring **yum** with the appropriate repository, you can now install the required **-devel**, **-debuginfo**, and **-debuginfo-common** packages for your kernel. To install the corresponding packages for a specific kernel, run the following commands:

- **yum install kernelname-devel-version**
- **debuginfo-install kernelname-version**

Replace **kernelname** with the appropriate kernel variant name (for example, **kernel-PAE**), and **version** with the target kernel's version. For example, to install the required kernel information packages for the **kernel-PAE-2.6.18-53.1.13.el5** kernel, run:

- **yum install kernel-PAE-devel-2.6.18-53.1.13.el5**
- **debuginfo-install kernel-PAE-2.6.18-53.1.13.el5**

If you do not have **yum** and **yum-utils** installed (and you are unable to install them), you will have to manually download and install the required kernel information packages. To generate the URL from which to download the required packages, use the following script:

fedoradebugurl.sh

```
#!/bin/bash
echo -n "Enter nvr of kernel-debuginfo (e.g. 2.6.25-14.fc9.x86_64) " ; \
read NVR; \
BASE=`uname -m` ; \
NVR=`echo $NVR | sed s/.$BASE//` ; \
VERSION=`echo $NVR | awk -F- '{print $1}'` ; \
RELEASE=`echo $NVR | awk -F- '{print $2}'` ; \
echo "http://kojipkgs.fedoraproject.org/\
packages/kernel/$VERSION/$RELEASE/$BASE/"
```

Once you have manually downloaded the required packages to the machine, install the RPMs by running **rpm --force -ivh *package_names***.

2.1.3. Initial Testing

If you are currently using the kernel you wish to probe with SystemTap, you can immediately test whether the deployment was successful. If not, you will need to reboot and load the appropriate kernel.

To start the test, run the command **stap -v -e 'probe vfs.read {printf("read performed\n"); exit()}'**. This command simply instructs SystemTap to print **read performed** then exit properly once a virtual file system read is detected. If the SystemTap deployment was successful, you should get output similar to the following:

```
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.
```

The last three lines of the output (i.e. beginning with **Pass 5** indicate that SystemTap was able to successfully create the instrumentation to probe the kernel, run the instrumentation, detect the event being probed (in this case, a virtual file system read), and execute a valid handler (print text then close it with no errors).

2.2. Generating Instrumentation for Other Computers

When users run a SystemTap script, SystemTap builds a kernel module out of that script. SystemTap then loads the module into the kernel, allowing it to extract the specified data directly from the kernel (refer to [Procedure 3.1, “SystemTap Session”](#) in [Section 3.1, “Architecture”](#) for more information).

Normally, however, SystemTap scripts can only be run on systems where SystemTap is deployed (as in [Section 2.1, “Installation and Setup”](#)). This could mean that if you want to run SystemTap on

ten systems, you would need to deploy SystemTap on *all* those systems. In some cases, this may be neither feasible nor desired. For instance, corporate policy may prohibit an administrator from installing RPMs that provide compilers or debug information on specific machines, which will prevent the deployment of SystemTap.

To work around this, you can resort to *cross-instrumentation*. Cross-instrumentation is the process of generating SystemTap instrumentation module from a SystemTap script on one computer to be used on another computer. This process offers the following benefits:

- The kernel information packages for various machines can be installed on a single *host machine*.
- Each *target machine* only needs one RPM to installed to use the generated SystemTap instrumentation module: **systemtap-runtime**.



Note

For the sake of simplicity, we will be using the following terms throughout this section:

- *instrumentation module* — the kernel module built from a SystemTap script; i.e. the *SystemTap module* is built on the *host system*, and will be loaded on the *target kernel* of *target system*.
- *host system* — the system on which you compile the instrumentation modules (from SystemTap scripts), to be loaded on *target systems*.
- *target system* — the system for which you are building the *instrumentation module* (from SystemTap scripts).
- *target kernel* — the kernel of the *target system*. This is the kernel on which you wish to load/run the *instrumentation module*.

Procedure 2.1. Configuring a Host System and Target Systems

1. Install the **systemtap-runtime** RPM on each *target system*.
2. Determine the kernel running on each *target system* by running **uname -r** on each *target system*.
3. Install SystemTap on the *host system*. You will be building the *instrumentation module* for the *target systems* on the *host system*. For instructions on how to install SystemTap, refer to [Section 2.1.1, “Installing SystemTap”](#).
4. Using the *target kernel* version determined earlier, install the *target kernel* and related RPMs on the *host system* by the method described in [Section 2.1.2, “Installing Required Kernel Information RPMs”](#). If multiple *target systems* use different *target kernels*, you will need to repeat this step for each different kernel used on the *target systems*.

After performing [Procedure 2.1, “Configuring a Host System and Target Systems”](#), you can now build the *instrumentation module* (for any *target system*) on the *host system*.

To build the *instrumentation module*, run the following command on the *host system* (be sure to specify the appropriate values):

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

Here, **kernel_version** refers to the version of the *target kernel* (the output of **uname -r** on the target machine), **script** refers to the script to be converted into an *instrumentation module*, and **module_name** is the desired name of the *instrumentation module*.



Note

To determine the architecture notation of a running kernel, run **uname -m**.

Once the *instrumentation module* is compiled, copy it to the *target system* and then load it using:

```
staprun module_name.ko
```

For example, to create the *instrumentation module* **simple.ko** from a SystemTap script named **simple.stp** for the *target kernel* 2.6.18-92.1.10.el5 (on x86_64 architecture), use the following command:

```
stap -r 2.6.18-92.1.10.el5 -e 'probe vfs.read {exit()}' -m simple
```

This will create a module named **simple.ko**. To use the *instrumentation module* **simple.ko**, copy it to the *target system* and run the following command (on the *target system*):

```
staprun simple.ko
```



Important

The *host system* must be the same architecture and running the same distribution of Linux as the *target system* in order for the built *instrumentation module* to work.

2.3. Running SystemTap Scripts

SystemTap scripts are run through the command **stap**. **stap** can run SystemTap scripts from standard input or from file.

Running **stap** and **staprun** requires elevated privileges to the system. However, not all users can be granted root access just to run SystemTap. In some cases, for instance, you may want to allow a non-privileged user to run SystemTap instrumentation on his machine.

To allow ordinary users to run SystemTap without root access, add them to one of these user groups:

stapdev

Members of this group can use **stap** to run SystemTap scripts, or **staprun** to run SystemTap instrumentation modules.

Running **stap** involves compiling SystemTap scripts into kernel modules and loading them into the kernel. This requires elevated privileges to the system, which are granted to **stapdev** members. Unfortunately, such privileges also grant effective root access to **stapdev** members.

As such, you should only grant **stapdev** group membership to users whom you can trust root access.

stapusr

Members of this group can only run **staprun** to run SystemTap instrumentation modules. In addition, they can only run those modules from **/lib/modules/kernel_version/systemtap/**. Note that this directory must be owned only by the root user, and must only be writable by the root user.

Below is a list of commonly used **stap** options:

-v

Makes the output of the SystemTap session more verbose. You can repeat this option (for example, **stap -vvv script.stp**) to provide more details on the script's execution. This option is particularly useful if you encounter any errors in running the script.

For more information about common SystemTap script errors, refer to [Chapter 5, Understanding SystemTap Errors](#).

-o filename

Sends the standard output to file (*filename*).

-x process ID

Sets the SystemTap handler function **target()** to the specified process ID. For more information about **target()**, refer to [SystemTap Functions](#).

-c command

Sets the SystemTap handler function **target()** to the specified command. Note that you must use the full path to the specified command; for example, instead of specifying **cp**, use **/bin/cp** (as in **stap script -c /bin/cp**). For more information about **target()**, refer to [SystemTap Functions](#).

-e 'script'

Use **script** string rather than a file as input for systemtap translator.

You can also instruct **stap** to run scripts from standard input using the switch **-**. To illustrate:

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

Example 2.1. Running Scripts From Standard Input

Example 2.1, "Running Scripts From Standard Input" instructs **stap** to run the script passed by **echo** to standard input. Any **stap** options you wish to use should be inserted before the **-** switch; for instance, to make the example in *Example 2.1, "Running Scripts From Standard Input"* more verbose, the command would be:

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

For more information about **stap**, refer to **man stap**.

To run SystemTap instrumentation (i.e. the kernel module built from SystemTap scripts during a cross-instrumentation), use **staprun** instead. For more information about **staprun** and cross-instrumentation, refer to [Section 2.2, "Generating Instrumentation for Other Computers"](#).

**Note**

The **stap** options **-v** and **-o** also work for **staprun**. For more information about **staprun**, refer to **man staprun**.

Understanding How SystemTap Works

SystemTap allows users to write and reuse simple scripts to deeply examine the activities of a running Linux system. These scripts can be designed to extract data, filter it, and summarize it quickly (and safely), enabling the diagnosis of complex performance (or even functional) problems.

The essential idea behind a SystemTap script is to name *events*, and to give them *handlers*. When SystemTap runs the script, SystemTap monitors for the event; once the event occurs, the Linux kernel then runs the handler as a quick sub-routine, then resumes.

There are several kind of events; entering/exiting a function, timer expiration, session termination, etc. A handler is a series of script language statements that specify the work to be done whenever the event occurs. This work normally includes extracting data from the event context, storing them into internal variables, and printing results.

3.1. Architecture

A SystemTap session begins when you run a SystemTap script. This session occurs in the following fashion:

Procedure 3.1. SystemTap Session

1. First, SystemTap checks the script against the existing tapset library (normally in `/usr/share/systemtap/tapset/` for any tapsets used. SystemTap will then substitute any located tapsets with their corresponding definitions in the tapset library.
2. SystemTap then translates the script to C, running the system C compiler to create a kernel module from it. The tools that perform this step are contained in the **systemtap** package (refer to [Section 2.1.1, “Installing SystemTap”](#) for more information).
3. SystemTap loads the module, then enables all the probes (events and handlers) in the script. The **staprun** in the **systemtap-runtime** package (refer to [Section 2.1.1, “Installing SystemTap”](#) for more information) provides this functionality.
4. As the events occur, their corresponding handlers are executed.
5. Once the SystemTap session is terminated, the probes are disabled, and the kernel module is unloaded.

This sequence is driven from a single command-line program: **stap**. This program is SystemTap's main front-end tool. For more information about **stap**, refer to **man stap** (once SystemTap is properly installed on your machine).

3.2. SystemTap Scripts

For the most part, SystemTap scripts are the foundation of each SystemTap session. SystemTap scripts instruct SystemTap on what type of information to collect, and what to do once that information is collected.

As stated in [Chapter 3, Understanding How SystemTap Works](#), SystemTap scripts are made up of two components: *events* and *handlers*. Once a SystemTap session is underway, SystemTap monitors the operating system for the specified events and executes the handlers as they occur.



Note

An event and its corresponding handler is collectively called a *probe*. A SystemTap script can have multiple probes.

A probe's handler is commonly referred to as a *probe body*.

In terms of application development, using events and handlers is similar to instrumenting the code by inserting diagnostic print statements in a program's sequence of commands. These diagnostic print statements allow you to view a history of commands executed once the program is run.

SystemTap scripts allow insertion of the instrumentation code without recompilation of the code and allows more flexibility with regard to handlers. Events serve as the triggers for handlers to run; handlers can be specified to record specified data and print it in a certain manner.

Format

SystemTap scripts use the file extension **.stp**, and contains probes written in the following format:

```
probe event {statements}
```

SystemTap supports multiple events per probe; multiple events are delimited by a comma (,). If multiple events are specified in a single probe, SystemTap will execute the handler when any of the specified events occur.

Each probe has a corresponding *statement block*. This statement block is enclosed in braces (**{ }**) and contains the statements to be executed per event. SystemTap executes these statements in sequence; special separators or terminators are generally not necessary between multiple statements.



Note

Statement blocks in SystemTap scripts follow the same syntax and semantics as the C programming language. A statement block can be nested within another statement block.

Systemtap allows you to write functions to factor out code to be used by a number of probes. Thus, rather than repeatedly writing the same series of statements in multiple probes, you can just place the instructions in a *function*, as in:

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

The **statements** in *function_name* are executed when the probe for *event* executes. The *arguments* are optional values passed into the function.



Important

Section 3.2, “SystemTap Scripts” is designed to introduce readers to the basics of SystemTap scripts. To understand SystemTap scripts better, it is advisable that you

refer to *Chapter 4, Useful SystemTap Scripts*; each section therein provides a detailed explanation of the script, its events, handlers, and expected output.

3.2.1. Event

SystemTap events can be broadly classified into two types: *synchronous* and *asynchronous*.

Synchronous Events

A *synchronous* event occurs when any process executes an instruction at a particular location in kernel code. This gives other events a reference point from which more contextual data may be available.

Examples of synchronous events include:

`syscall.system_call`

The entry to the system call `system_call`. If the exit from a syscall is desired, appending a `.return` to the event monitor the exit of the system call instead. For example, to specify the entry and exit of the system call `close`, use `syscall.close` and `syscall.close.return` respectively.

`vfs.file_operation`

The entry to the `file_operation` event for Virtual File System (VFS). Similar to `syscall` event, appending a `.return` to the event monitors the exit of the `file_operation` operation.

`kernel.function("function")`

The entry to the kernel function `function`. For example, `kernel.function("sys_open")` refers to the "event" that occurs when the kernel function `sys_open` is called by any thread in the system. To specify the *return* of the kernel function `sys_open`, append the `return` string to the event statement; i.e. `kernel.function("sys_open").return`.

When defining probe events, you can use asterisk (*) for wildcards. You can also trace the entry or exit of a function in a kernel source file. Consider the following example:

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

Example 3.1. wildcards.stp

In the previous example, the first probe's event specifies the entry of ALL functions in the kernel source file `net/socket.c`. The second probe specifies the exit of all those functions. Note that in this example, there are no statements in the handler; as such, no information will be collected or displayed.

`kernel.trace("tracepoint")`

The static probe for `tracepoint`. Recent kernels (2.6.30 and newer) include instrumentation for specific events in the kernel. These events are statically marked with tracepoints. One example of a tracepoint available in systemtap is `kernel.trace("kfree_skb")` which indicates each time a network buffer is freed in the kernel.

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

Allows you to probe functions within modules. For example:

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

Example 3.2. moduleprobe.stp

The first probe in [Example 3.2, “moduleprobe.stp”](#) points to the entry of *all* functions for the **ext3** module. The second probe points to the exits of all functions for that same module; the use of the **.return** suffix is similar to **kernel.function()**. Note that the probes in [Example 3.2, “moduleprobe.stp”](#) do not contain statements in the probe handlers, and as such will not print any useful data (as in [Example 3.1, “wildcards.stp”](#)).

A system's kernel modules are typically located in **/lib/modules/kernel_version**, where *kernel_version* refers to the currently loaded kernel version. Modules use the filename extension **.ko**.

Asynchronous Events

Asynchronous events are not tied to a particular instruction or location in code. This family of probe points consists mainly of counters, timers, and similar constructs.

Examples of asynchronous events include:

begin

The startup of a SystemTap session; i.e. as soon as the SystemTap script is run.

end

The end of a SystemTap session.

timer events

An event that specifies a handler to be executed periodically. For example:

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

Example 3.3. timer-s.stp

[Example 3.3, “timer-s.stp”](#) is an example of a probe that prints **hello world** every 4 seconds. Note that you can also use the following timer events:

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

When used in conjunction with other probes that collect information, timer events allows you to print out get periodic updates and see how that information changes over time.



Important

SystemTap supports the use of a large collection of probe events. For more information about supported events, refer to `man stapprobes`. The *SEE ALSO* section of `man stapprobes` also contains links to other `man` pages that discuss supported events for specific subsystems and components.

3.2.2. Systemtap Handler/Body

Consider the following sample script:

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

Example 3.4. `helloworld.stp`

In [Example 3.4, “`helloworld.stp`”](#), the event **begin** (i.e. the start of the session) triggers the handler enclosed in `{ }`, which simply prints **hello world** followed by a new-line, then exits.



Note

SystemTap scripts continue to run until the `exit()` function executes. If the users wants to stop the execution of the script, it can interrupted manually with **Ctrl+C**.

`printf ()` Statements

The `printf ()` statement is one of the simplest functions for printing data. `printf ()` can also be used to display data using a wide variety of SystemTap functions in the following format:

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

The *format string* specifies how *arguments* should be printed. The format string of [Example 3.4, “`helloworld.stp`”](#) simply instructs SystemTap to print **hello world**, and contains no format specifiers.

You can use the format specifiers `%s` (for strings) and `%d` (for numbers) in format strings, depending on your list of arguments. Format strings can have multiple format specifiers, each matching a corresponding argument; multiple arguments are delimited by a comma (,).



Note

Semantically, the SystemTap `printf` function is very similar to its C language counterpart. The aforementioned syntax and format for SystemTap's `printf` function is identical to that of the C-style `printf`.

To illustrate this, consider the following probe example:

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

[Example 3.5. variables-in-printf-statements.stp](#)

Example 3.5, “variables-in-printf-statements.stp” instructs SystemTap to probe all entries to the system call **open**; for each event, it prints the current **execname()** (a string with the executable name) and **pid()** (the current process ID number), followed by the word **open**. A snippet of this probe's output would look like:

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

SystemTap Functions

SystemTap supports a wide variety of functions that can be used as **printf()** arguments.

Example 3.5, “variables-in-printf-statements.stp” uses the SystemTap functions **execname()** (name of the process that called a kernel function/performed a system call) and **pid()** (current process ID).

The following is a list of commonly-used SystemTap functions:

tid()

The ID of the current thread.

uid()

The ID of the current user.

cpu()

The current CPU number.

gettimeofday_s()

The number of seconds since UNIX epoch (January 1, 1970).

ctime()

Convert number of seconds since UNIX epoch to date.

pp()

A string describing the probe point currently being handled.

thread_indent()

This particular function is quite useful, providing you with a way to better organize your print results. The function takes one argument, an indentation delta, which indicates how many spaces to add or remove from a thread's "indentation counter". It then returns a string with some generic trace data along with an appropriate number of indentation spaces.

The generic data included in the returned string includes a timestamp (number of microseconds since the first call to **thread_indent()** by the thread), a process name, and the thread ID. This allows you to identify what functions were called, who called them, and the duration of each function call.

If call entries and exits immediately precede each other, it is easy to match them. However, in most cases, after a first function call entry is made several other call entries and exits may be made before the first call exits. The indentation counter helps you match an entry with its corresponding exit by indenting the next function call if it is not the exit of the previous one.

Consider the following example on the use of **thread_indent()**:

```
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())
}
```

Example 3.6. *thread_indent.stp*

Example 3.6, “*thread_indent.stp*” prints out the **thread_indent()** and probe functions at each event in the following format:

```
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
```

This sample output contains the following information:

- The time (in microseconds) since the initial **thread_indent()** call for the thread (included in the string from **thread_indent()**).
- The process name (and its corresponding ID) that made the function call (included in the string from **thread_indent()**).
- An arrow signifying whether the call was an entry (<-) or an exit (->); the indentations help you match specific function call entries with their corresponding exits.
- The name of the function called by the process.

name

Identifies the name of a specific system call. This variable can only be used in probes that use the event **syscall.system_call**.

target()

Used in conjunction with **stap script -x process ID** or **stap script -c command**. If you want to specify a script to take an argument of a process ID or command, use **target()** as the variable in the script to refer to it. For example:

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

[Example 3.7. targetexample.stp](#)

When [Example 3.7, “targetexample.stp”](#) is run with the argument **-x process ID**, it watches all system calls (as specified by the event **syscall.***) and prints out the name of all system calls made by the specified process.

This has the same effect as specifying **if (pid() == process ID)** each time you wish to target a specific process. However, using **target()** makes it easier for you to re-use the script, giving you the ability to simply pass a process ID as an argument each time you wish to run the script (e.g. **stap targetexample.stp -x process ID**).

For more information about supported SystemTap functions, refer to **man stapfuncs**.

3.3. Basic SystemTap Handler Constructs

SystemTap supports the use of several basic constructs in handlers. The syntax for most of these handler constructs are mostly based on C and **awk** syntax. This section describes several of the most useful SystemTap handler constructs, which should provide you with enough information to write simple yet useful SystemTap scripts.

3.3.1. Variables

Variables can be used freely throughout a handler; simply choose a name, assign a value from a function or expression to it, and use it in an expression. SystemTap automatically identifies whether a variable should be typed as a string or integer, based on the type of the values assigned to it. For instance, if you use set the variable **foo** to **gettimeofday_s()** (as in **foo = gettimeofday_s()**), then **foo** is typed as an number and can be printed in a **printf()** with the integer format specifier (**%d**).

Note, however, that by default variables are only local to the probe they are used in. This means that variables are initialized, used and disposed at each probe handler invocation. To share a variable between probes, declare the variable name using **global** outside of the probes. Consider the following example:

```
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 ()
}
```

Example 3.8. timer-jiffies.stp

Example 3.8, “timer-jiffies.stp” computes the **CONFIG_HZ** setting of the kernel using timers that count jiffies and milliseconds, then computing accordingly. The **global** statement allows the script to use the variables **count_jiffies** and **count_ms** (set in their own respective probes) to be shared with **probe timer.ms(12345)**.

Note

The ++ notation in *Example 3.8, “timer-jiffies.stp”* (i.e. **count_jiffies ++** and **count_ms ++**) is used to increment the value of a variable by 1. In the following probe, **count_jiffies** is incremented by 1 every 100 jiffies:

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

In this instance, SystemTap understands that **count_jiffies** is an integer. Because no initial value was assigned to **count_jiffies**, its initial value is zero by default.

3.3.2. Conditional Statements

In some cases, the output of a SystemTap script may be too big. To address this, you need to further refine the script's logic in order to delimit the output into something more relevant or useful to your probe.

You can do this by using *conditionals* in handlers. SystemTap accepts the following types of conditional statements:

If/Else Statements

Format:

```
if (condition)
  statement1
else
  statement2
```

The **statement1** is executed if the **condition** expression is non-zero. The **statement2** is executed if the **condition** expression is zero. The **else** clause (**else statement2**) is optional. Both **statement1** and **statement2** can be statement blocks.

```
global countread, countnonread
probe kernel.function("vfs_read"), kernel.function("vfs_write")
{
  if (probfunc()=="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)
}
```

Example 3.9. ifelse.stp

Example 3.9, “ifelse.stp” is a script that counts how many virtual file system reads (**vfs_read**) and writes (**vfs_write**) the system performs within a 5-second span. When run, the script increments the value of the variable **countread** by 1 if the name of the function it probed matches **vfs_read** (as noted by the condition **if (probefunc()=="vfs_read")**); otherwise, it increments **countnonread** (**else {countnonread ++}**).

While Loops

Format:

```
while (condition)
    statement
```

So long as **condition** is non-zero the block of statements in **statement** are executed. The **statement** is often a statement block and it must change a value so **condition** will eventually be zero.

For Loops

Format:

```
for (initialization; conditional; increment) statement
```

The **for** loop is simply shorthand for a while loop. The following is the equivalent **while** loop:

```
initialization
while (conditional) {
    statement
    increment
}
```

Conditional Operators

Aside from **==** (“is equal to”), you can also use the following operators in your conditional statements:

>=

Greater than or equal to

<=

Less than or equal to

!=

Is not equal to

3.3.3. Command-Line Arguments

You can also allow a SystemTap script to accept simple command-line arguments using a **\$** or **@** immediately followed by the number of the argument on the command line. Use **\$** if you are expecting the user to enter an integer as a command-line argument, and **@** if you are expecting a string.

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

Example 3.10. `commandlineargs.stp`

Example 3.10, “`commandlineargs.stp`” is similar to *Example 3.1, “`wildcards.stp`”*, except that it allows you to pass the kernel function to be probed as a command-line argument (as in **stap commandlineargs.stp kernel function**). You can also specify the script to accept multiple command-line arguments, noting them as **@1**, **@2**, and so on, in the order they are entered by the user.

3.4. Associative Arrays

SystemTap also supports the use of associative arrays. While an ordinary variable represents a single value, associative arrays can represent a collection of values. Simply put, an associative array is a collection of unique keys; each key in the array has a value associated with it.

Since associative arrays are normally processed in multiple probes (as we will demonstrate later), they are declared as **global** variables in the SystemTap script. The syntax for accessing an element in an associative array is similar to that of **awk**, and is as follows:

```
array_name[index_expression]
```

Here, the **array_name** is any arbitrary name the array uses. The **index_expression** is used to refer to a specific unique key in the array. To illustrate, let us try to build an array named **foo** that specifies the ages of three people (i.e. the unique keys): **tom**, **dick**, and **harry**. To assign them the ages (i.e. associated values) of 23, 24, and 25 respectively, we'd use the following array statements:

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

Example 3.11. Basic Array Statements

You can specify up to 5 index expressions in an array statement, each one delimited by a comma (,). This is useful if you wish to have a key that contains multiple pieces of information. The following line from *disktop.stp* uses 5 elements for the key: process ID, executable name, user ID, parent process ID, and string "W". It associates the value of **devname** with that key.

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



Important

All associate arrays must be declared as **global**, regardless of whether the associate array is used in one or multiple probes.

3.5. Array Operations in SystemTap

This section enumerates some of the most commonly used array operations in SystemTap.

3.5.1. Assigning an Associated Value

Use `=` to set an associated value to indexed unique pairs, as in:

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

Example 3.11, “Basic Array Statements” shows a very basic example of how to set an explicit associated value to a unique key. You can also use a handler function as both your **index_expression** and **value**. For example, you can use arrays to set a timestamp as the associated value to a process name (which you wish to use as your unique key), as in:

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

Example 3.12. Associating Timestamps to Process Names

Whenever an event invokes the statement in *Example 3.12, “Associating Timestamps to Process Names”*, SystemTap returns the appropriate **tid()** value (i.e. the ID of a thread, which is then used as the unique key). At the same time, SystemTap also uses the function **gettimeofday_s()** to set the corresponding timestamp as the associated value to the unique key defined by the function **tid()**. This creates an array composed of key pairs containing thread IDs and timestamps.

In this same example, if **tid()** returns a value that is already defined in the array **foo**, the operator will discard the original associated value to it, and replace it with the current timestamp from **gettimeofday_s()**.

3.5.2. Reading Values From Arrays

You can also read values from an array the same way you would read the value of a variable. To do so, include the **array_name[index_expression]** statement as an element in a mathematical expression. For example:

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

Example 3.13. Using Array Values in Simple Computations

This example assumes that the array **foo** was built using the construct in *Example 3.12, “Associating Timestamps to Process Names”* (from *Section 3.5.1, “Assigning an Associated Value”*). This sets a timestamp that will serve as a *reference point*, to be used in computing for **delta**.

The construct in [Example 3.13, “Using Array Values in Simple Computations”](#) computes a value for the variable **delta** by subtracting the associated value of the key **tid()** from the current **gettimeofday_s()**. The construct does this by *reading* the value of **tid()** from the array. This particular construct is useful for determining the time between two events, such as the start and completion of a read operation.



Note

If the *index_expression* cannot find the unique key, it returns a value of 0 (for numerical operations, such as [Example 3.13, “Using Array Values in Simple Computations”](#)) or a null/empty string value (for string operations) by default.

3.5.3. Incrementing Associated Values

Use **++** to increment the associated value of a unique key in an array, as in:

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

Again, you can also use a handler function for your *index_expression*. For example, if you wanted to tally how many times a specific process performed a read to the virtual file system (using the event **vfs.read**), you can use the following probe:

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

[Example 3.14. vfsreads.stp](#)

In [Example 3.14, “vfsreads.stp”](#), the first time that the probe returns the process name **gnome-terminal** (i.e. the first time **gnome-terminal** performs a VFS read), that process name is set as the unique key **gnome-terminal** with an associated value of 1. The next time that the probe returns the process name **gnome-terminal**, SystemTap increments the associated value of **gnome-terminal** by 1. SystemTap performs this operation for *all* process names as the probe returns them.

3.5.4. Processing Multiple Elements in an Array

Once you've collected enough information in an array, you will need to retrieve and process all elements in that array to make it useful. Consider [Example 3.14, “vfsreads.stp”](#): the script collects information about how many VFS reads each process performs, but does not specify what to do with it. The obvious means for making [Example 3.14, “vfsreads.stp”](#) useful is to print the key pairs in the array **reads**, but how?

The best way to process all key pairs in an array (as an iteration) is to use the **foreach** statement. Consider the following example:

```
global reads
probe vfs.read
```

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

Example 3.15. *cumulative-vfsreads.stp*

In the second probe of [Example 3.15, “*cumulative-vfsreads.stp*”](#), the **foreach** statement uses the variable **count** to reference each iteration of a unique key in the array **reads**. The **reads[count]** array statement in the same probe retrieves the associated value of each unique key.

Given what we know about the first probe in [Example 3.15, “*cumulative-vfsreads.stp*”](#), the script prints VFS-read statistics every 3 seconds, displaying names of processes that performed a VFS-read along with a corresponding VFS-read count.

Now, remember that the **foreach** statement in [Example 3.15, “*cumulative-vfsreads.stp*”](#) prints *all* iterations of process names in the array, and in no particular order. You can instruct the script to process the iterations in a particular order by using **+** (ascending) or **-** (descending). In addition, you can also limit the number of iterations the script needs to process with the **limit value** option.

For example, consider the following replacement probe:

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

This **foreach** statement instructs the script to process the elements in the array **reads** in descending order (of associated value). The **limit 10** option instructs the **foreach** to only process the first ten iterations (i.e. print the first 10, starting with the highest value).

3.5.5. Clearing/Deleting Arrays and Array Elements

Sometimes, you may need to clear the associated values in array elements, or reset an entire array for re-use in another probe. [Example 3.15, “*cumulative-vfsreads.stp*”](#) in [Section 3.5.4, “*Processing Multiple Elements in an Array*”](#) allows you to track how the number of VFS reads per process grows over time, but it does not show you the number of VFS reads each process makes per 3-second period.

To do that, you will need to clear the values accumulated by the array. You can accomplish this using the **delete** operator to delete elements in an array, or an entire array. Consider the following example:

```
global reads
probe vfs.read
{
  reads[execname()] ++
}
probe timer.s(3)
```



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

Example 3.16. *noncumulative-vfsreads.stp*

In [Example 3.16](#), “*noncumulative-vfsreads.stp*”, the second probe prints the number of VFS reads each process made *within the probed 3-second period only*. The **delete reads** statement clears the **reads** array within the probe.



Note

You can have multiple array operations within the same probe. Using the examples from [Section 3.5.4](#), “*Processing Multiple Elements in an Array*” and [Section 3.5.5](#), “*Clearing/Deleting Arrays and Array Elements*”, you can track the number of VFS reads each process makes per 3-second period *and* tally the cumulative VFS reads of those same processes. Consider the following example:

```
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])
}
```

In this example, the arrays **reads** and **totalreads** track the same information, and are printed out in a similar fashion. The only difference here is that **reads** is cleared every 3-second period, whereas **totalreads** keeps growing.

3.5.6. Using Arrays in Conditional Statements

You can also use associative arrays in **if** statements. This is useful if you want to execute a subroutine once a value in the array matches a certain condition. Consider the following example:

```
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])
}
```

Example 3.17. *vfsreads-print-if-1kb.stp*

Every three seconds, [Example 3.17](#), “*vfsreads-print-if-1kb.stp*” prints out a list of all processes, along with how many times each process performed a VFS read. If the associated value of a process name is equal or greater than 1024, the **if** statement in the script converts and prints it out in **kB**.

Testing for Membership

You can also test whether a specific unique key is a member of an array. Further, membership in an array can be used in **if** statements, as in:

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

To illustrate this, consider the following example:

```
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()
    }
}
```

Example 3.18. *vfsreads-stop-on-stapio2.stp*

The **if(["stapio"] in reads)** statement instructs the script to print **stapio read detected, exiting** once the unique key **stapio** is added to the array **reads**.

3.5.7. Computing for Statistical Aggregates

Statistical aggregates are used to collect statistics on numerical values where it is important to accumulate new data quickly and in large volume (i.e. storing only aggregated stream statistics). Statistical aggregates can be used in global variables or as elements in an array.

To add value to a statistical aggregate, use the operator `<<< value`.

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

Example 3.19. stat-aggregates.stp

In [Example 3.19, “stat-aggregates.stp”](#), the operator `<<< count` stores the amount returned by `count` to the associated value of the corresponding `execname()` in the `reads` array. Remember, these values are *stored*; they are not added to the associated values of each unique key, nor are they used to replace the current associated values. In a manner of speaking, think of it as having each unique key (`execname()`) having multiple associated values, accumulating with each probe handler run.



Note

In the context of [Example 3.19, “stat-aggregates.stp”](#), `count` returns the amount of data written by the returned `execname()` to the virtual file system.

To extract data collected by statistical aggregates, use the syntax format `@extractor(variable/array index expression)`. `extractor` can be any of the following integer extractors:

count

Returns the number of all values stored into the variable/array index expression. Given the sample probe in [Example 3.19, “stat-aggregates.stp”](#), the expression `@count(writes[execname()])` will return *how many values are stored in each unique key in array writes*.

sum

Returns the sum of all values stored into the variable/array index expression. Again, given sample probe in [Example 3.19, “stat-aggregates.stp”](#), the expression `@sum(writes[execname()])` will return *the total of all values stored in each unique key in array writes*.

min

Returns the smallest among all the values stored in the variable/array index expression.

max

Returns the largest among all the values stored in the variable/array index expression.

avg

Returns the average of all values stored in the variable/array index expression.

When using statistical aggregates, you can also build array constructs that use multiple index expressions (to a maximum of 5). This is helpful in capturing additional contextual information during a probe. For example:

```
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]))
}
```

Example 3.20. Multiple Array Indexes

In [Example 3.20, “Multiple Array Indexes”](#), the first probe tracks how many times each process performs a VFS read. What makes this different from earlier examples is that this array associates a performed read to both a process name *and* its corresponding process ID.

The second probe in [Example 3.20, “Multiple Array Indexes”](#) demonstrates how to process and print the information collected by the array **reads**. Note how the **foreach** statement uses the same number of variables (i.e. **var1** and **var2**) contained in the first instance of the array **reads** from the first probe.

3.6. Tapsets

Tapsets are scripts that form a library of pre-written probes and functions to be used in SystemTap scripts. When a user runs a SystemTap script, SystemTap checks the script's probe events and handlers against the tapset library; SystemTap then loads the corresponding probes and functions before translating the script to C (refer to [Section 3.1, “Architecture”](#) for information on what transpires in a SystemTap session).

Like SystemTap scripts, tapsets use the filename extension **.stp**. The standard library of tapsets is located in **/usr/share/systemtap/tapset/** by default. However, unlike SystemTap scripts, tapsets are not meant for direct execution; rather, they constitute the library from which other scripts can pull definitions.

Simply put, the tapset library is an abstraction layer designed to make it easier for users to define events and functions. In a manner of speaking, tapsets provide useful aliases for functions that users may want to specify as an event; knowing the proper alias to use is, for the most part, easier than remembering specific kernel functions that might vary between kernel versions.

Several handlers and functions in [Section 3.2.1, “Event”](#) and [SystemTap Functions](#) are defined in tapsets. For example, **thread_indent()** is defined in **indent.stp**.

Useful SystemTap Scripts

This chapter enumerates several SystemTap scripts you can use to monitor and investigate different subsystems. All of these scripts are available at `/usr/share/systemtap/testsuite/systemtap.examples/` once you install the **systemtap-testsuite** RPM.

4.1. Network

The following sections showcase scripts that trace network-related functions and build a profile of network activity.

4.1.1. Network Profiling

This section describes how to profile network activity. [nettop.stp](#) provides a glimpse into how much network traffic each process is generating on a machine.

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 %-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()
}
```

Note that **function** `print_activity()` uses the following expressions:

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

These expressions are if/else conditionals. The first statement is simply a more concise way of writing the following psuedo code:

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

[*nettop.stp*](#) tracks which processes are generating network traffic on the system, and provides the following information about each process:

- **PID** — the ID of the listed process.
- **UID** — user ID. A user ID of **0** refers to the root user.
- **DEV** — which ethernet device the process used to send / receive data (e.g. eth0, eth1)
- **XMIT_PK** — number of packets transmitted by the process
- **RECV_PK** — number of packets received by the process
- **XMIT_KB** — amount of data sent by the process, in kilobytes
- **RECV_KB** — amount of data received by the service, in kilobytes

[*nettop.stp*](#) provides network profile sampling every 5 seconds. You can change this setting by editing **probe timer.ms(5000)** accordingly. [Example 4.1, “nettop.stp Sample Output”](#) contains an excerpt of the output from [*nettop.stp*](#) over a 20-second period:

```
[...]
```

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-poll
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-poll
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
[...]
```

Example 4.1. *nettop.stp* Sample Output

4.1.2. Tracing Functions Called in Network Socket Code

This section describes how to trace functions called from the kernel's **net/socket.c** file. This task helps you identify, in finer detail, how each process interacts with the network at the kernel level.

socket-trace.stp

```

#!/usr/bin/env 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())
}
```

socket-trace.stp is identical to [Example 3.6, “thread_indent.stp”](#), which was earlier used in [SystemTap Functions](#) to illustrate how **thread_indent()** works.

```

[...]
```

```

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
```

[...]

Example 4.2. *socket-trace.stp* Sample Output

Example 4.2, “*socket-trace.stp* Sample Output” contains a 3-second excerpt of the output for *socket-trace.stp*. For more information about the output of this script as provided by `thread_indent()`, refer to *SystemTap Functions Example 3.6, “thread_indent.stp”*.

4.1.3. Monitoring Incoming TCP Connections

This section illustrates how to monitor incoming TCP connections. This task is useful in identifying any unauthorized, suspicious, or otherwise unwanted network access requests in real time.

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))
}
```

While *tcp_connections.stp* is running, it will print out the following information about any incoming TCP connections accepted by the system in real time:

- Current **UID**
- **CMD** - the command accepting the connection
- **PID** of the command
- Port used by the connection
- IP address from which the TCP connection originated

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

Example 4.3. *tcp_connections.stp* Sample Output

4.1.4. Monitoring TCP Packets

This section illustrates how to monitor TCP packets received by the system. This is useful in analyzing network traffic generated by applications running on the system.

tcpdumplike.stp

```

#!/usr/bin/env stp

// A TCP dump like example

probe begin, timer.s(1) {
    printf("-----\n")
    printf("      Source IP      Dest IP  SPort  DPort  U  A  P  R  S  F \n")
    printf("-----\n")
}

probe tcp.receive {
    printf(" %15s %15s %5d %5d %d %d %d %d %d %d\n",
        saddr, daddr, sport, dport, urg, ack, psh, rst, syn, fin)
}

```

While [tcpdumplike.stp](#) is running, it will print out the following information about any received TCP packets in real time:

- Source and destination IP address (**saddr**, **daddr**, respectively)
- Source and destination ports (**sport**, **dport**, respectively)
- Packet flags

To determine the flags used by the packet, [tcpdumplike.stp](#) uses the following functions:

- **urg** - urgent
- **ack** - acknowledgement
- **psh** - push
- **rst** - reset
- **syn** - synchronize
- **fin** - finished

The aforementioned functions return **1** or **0** to specify whether the packet uses the corresponding flag.

Source IP	Dest IP	SPort	DPort	U	A	P	R	S	F
209.85.229.147	10.0.2.15	80	20373	0	1	1	0	0	0
92.122.126.240	10.0.2.15	80	53214	0	1	0	0	1	0
92.122.126.240	10.0.2.15	80	53214	0	1	0	0	0	0
209.85.229.118	10.0.2.15	80	63433	0	1	0	0	1	0
209.85.229.118	10.0.2.15	80	63433	0	1	0	0	0	0
209.85.229.147	10.0.2.15	80	21141	0	1	1	0	0	0
209.85.229.147	10.0.2.15	80	21141	0	1	1	0	0	0
209.85.229.147	10.0.2.15	80	21141	0	1	1	0	0	0
209.85.229.147	10.0.2.15	80	21141	0	1	1	0	0	0
209.85.229.147	10.0.2.15	80	21141	0	1	1	0	0	0
209.85.229.147	10.0.2.15	80	21141	0	1	1	0	0	0
209.85.229.118	10.0.2.15	80	63433	0	1	1	0	0	0

```
[...]
```

Example 4.4. *tcpdumplike.stp* Sample Output

4.1.5. Monitoring Network Packets Drops in Kernel

The network stack in Linux can discard packets for various reasons. Some Linux kernels include a tracepoint, **kernel.trace("kfree_skb")**, which easily tracks where packets are discarded. *dropwatch.stp* uses **kernel.trace("kfree_skb")** to trace packet discards; the script summarizes which locations discard packets every five-second interval.

dropwatch.stp

```
#!/usr/bin/env 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
}
```

The **kernel.trace("kfree_skb")** traces which places in the kernel drop network packets. The **kernel.trace("kfree_skb")** has two arguments: a pointer to the buffer being freed (**\$skb**) and the location in kernel code the buffer is being freed (**\$location**).

Running the *dropwatch.stp* script 15 seconds would result in output similar in [Example 4.5, "dropwatch.stp Sample Output"](#). The output lists the number of misses for tracepoint address and the actual address.

```
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

```

Example 4.5. *dropwatch.stp* Sample Output

To make the location of packet drops more meaningful, refer to the `/boot/System.map-`uname -r`` file. This file lists the starting addresses for each function, allowing you to map the addresses in the output of [Example 4.5, “dropwatch.stp Sample Output”](#) to a specific function name. Given the following snippet of the `/boot/System.map-`uname -r`` file, the address `0xffffffff8024cd0f` maps to the function `unix_stream_recvmsg` and the address `0xffffffff8044b472` maps to the function `arp_rcv`:

```

[...]
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. Disk

The following sections showcase scripts that monitor disk and I/O activity.

4.2.1. Summarizing Disk Read/Write Traffic

This section describes how to identify which processes are performing the heaviest disk reads/writes to the system.

disktop.stp

```

#!/usr/bin/env 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%4dB/sec, %-7s%6dB, %-7s%6dB\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
}
```

[*disktop.stp*](#) outputs the top ten processes responsible for the heaviest reads/writes to disk.

[*Example 4.6, “disktop.stp Sample Output”*](#) displays a sample output for this script, and includes the following data per listed process:

- **UID** — user ID. A user ID of **0** refers to the root user.
- **PID** — the ID of the listed process.

- **PPID** — the process ID of the listed process's *parent process*.
- **CMD** — the name of the listed process.
- **DEVICE** — which storage device the listed process is reading from or writing to.
- **T** — the type of action performed by the listed process; **W** refers to write, while **R** refers to read.
- **BYTES** — the amount of data read to or written from disk.

The time and date in the output of [disktop.stp](#) is returned by the functions `ctime()` and `gettimeofday_s()`. `ctime()` derives calendar time in terms of seconds passed since the Unix epoch (January 1, 1970). `gettimeofday_s()` counts the *actual* number of seconds since Unix epoch, which gives a fairly accurate human-readable timestamp for the output.

In this script, the `$return` is a local variable that stores the actual number of bytes each process reads or writes from the virtual file system. `$return` can only be used in return probes (e.g. `vfs.read.return` and `vfs.read.return`).

```
[...]
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
```

Example 4.6. [disktop.stp](#) Sample Output

4.2.2. Tracking I/O Time For Each File Read or Write

This section describes how to monitor the amount of time it takes for each process to read from or write to any file. This is useful if you wish to determine what files are slow to load on a given system.

[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]
}

```

iotime.stp tracks each time a system call opens, closes, reads from, and writes to a file. For each file any system call accesses, *iotime.stp* counts the number of microseconds it takes for any reads or writes to finish and tracks the amount of data (in bytes) read from or written to the file.

iotime.stp also uses the local variable **\$count** to track the amount of data (in bytes) that any system call *attempts* to read or write. Note that **\$return** (as used in *disktop.stp* from [Section 4.2.1, “Summarizing Disk Read/Write Traffic”](#)) stores the *actual* amount of data read/written. **\$count** can only be used on probes that track data reads or writes (e.g. **syscall.read** and **syscall.write**).

```

[...]
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
[...]

```

Example 4.7. *iotime.stp* Sample Output

Example 4.7, “iotime.stp Sample Output” prints out the following data:

- A timestamp, in microseconds
- Process ID and process name
- An **access** or **iotime** flag
- The file accessed

If a process was able to read or write any data, a pair of **access** and **iotime** lines should appear together. The **access** line's timestamp refer to the time that a given process started accessing a file; at the end of the line, it will show the amount of data read/written (in bytes). The **iotime** line will show the amount of time (in microseconds) that the process took in order to perform the read or write.

If an **access** line is not followed by an **iotime** line, it simply means that the process did not read or write any data.

4.2.3. Track Cumulative IO

This section describes how to track the cumulative amount of I/O to the system.

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.
}
```

[traceio.stp](#) prints the top ten executables generating I/O traffic over time. In addition, it also tracks the cumulative amount of I/O reads and writes done by those ten executables. This information is tracked and printed out in 1-second intervals, and in descending order.

Note that [traceio.stp](#) also uses the local variable `$return`, which is also used by [disktop.stp](#) from [Section 4.2.1, “Summarizing Disk Read/Write Traffic”](#).

```
[...]
      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
      pcsd 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

```

Example 4.8. *traceio.stp* Sample Output

4.2.4. I/O Monitoring (By Device)

This section describes how to monitor I/O activity on a specific device.

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)
}

```

traceio2.stp takes 1 argument: the whole device number. To get this number, use **stat -c "0x%D" *directory***, where *directory* is located in the device you wish to monitor.

The **usrdev2kerndev()** function converts the whole device number into the format understood by the kernel. The output produced by **usrdev2kerndev()** is used in conjunction with the **MKDEV()**, **MINOR()**, and **MAJOR()** functions to determine the major and minor numbers of a specific device.

The output of *traceio2.stp* includes the name and ID of any process performing a read/write, the function it is performing (i.e. **vfs_read** or **vfs_write**), and the kernel device number.

The following example is an excerpt from the full output of **stap traceio2.stp 0x805**, where **0x805** is the whole device number of **/home**. **/home** resides in **/dev/sda5**, which is the device we wish to monitor.

```
[...]
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
[...]
```

Example 4.9. *traceio2.stp* Sample Output

4.2.5. Monitoring Reads and Writes to a File

This section describes how to monitor reads from and writes to a file in real time.

inodewatch-simple.stp

```
probe vfs.write, vfs.read
{
    dev_nr = $file->f_dentry->d_inode->i_sb->s_dev
    inode_nr = $file->f_dentry->d_inode->i_ino
    if (dev_nr == ($1 << 20 | $2) # major/minor device
        && inode_nr == $3)
        printf ("%s(%d) %s 0x%x/%u\n",
            execname(), pid(), probefunc(), dev_nr, inode_nr)
}
```

inodewatch-simple.stp takes the following information about the file as arguments on the command line:

- The file's major device number.
- The file's minor device number.
- The file's **inode** number.

To get this information, use **stat -c '%D %i' *filename***, where *filename* is an absolute path.

For instance: if you wish to monitor **/etc/crontab**, run **stat -c '%D %i' /etc/crontab** first. This gives the following output:

```
805 1078319
```

805 is the base-16 (hexadecimal) device number. The lower two digits are the minor device number and the upper digits are the major number. **1078319** is the **inode** number. To start monitoring **/etc/crontab**, run **stap inodewatch.stp 0x8 0x05 1078319** (The **0x** prefixes indicate base-16 values).

The output of this command contains the name and ID of any process performing a read/write, the function it is performing (i.e. **vfs_read** or **vfs_write**), the device number (in hex format), and the **inode** number. [Example 4.10, “inodewatch-simple.stp Sample Output”](#) contains the output of **stap inodewatch.stp 0x8 0x05 1078319** (when **cat /etc/crontab** is executed while the script is running) :

```
cat(16437) vfs_read 0x8000005/1078319
cat(16437) vfs_read 0x8000005/1078319
```

Example 4.10. *inodewatch-simple.stp* Sample Output

4.2.6. Monitoring Changes to File Attributes

This section describes how to monitor if any processes are changing the attributes of a targeted file, in real time.

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())
}
```

Like *inodewatch-simple.stp* from [Section 4.2.5, “Monitoring Reads and Writes to a File”](#), *inodewatch2-simple.stp* takes the targeted file's device number (in integer format) and **inode** number as arguments. For more information on how to retrieve this information, refer to [Section 4.2.5, “Monitoring Reads and Writes to a File”](#).

The output for *inodewatch2-simple.stp* is similar to that of *inodewatch-simple.stp*, except that *inodewatch2-simple.stp* also contains the attribute changes to the monitored file, as well as the ID of the user responsible (**uid()**). [Example 4.11, “inodewatch2-simple.stp Sample Output”](#) contains shows the output of *inodewatch2-simple.stp* while monitoring **/home/joe/bigfile** when user **joe** executes **chmod 777 /home/joe/bigfile** and **chmod 666 /home/joe/bigfile**.

```
chmod(17448) inode_setattr 0x8000005/6011835 100777 500
chmod(17449) inode_setattr 0x8000005/6011835 100666 500
```

Example 4.11. *inodewatch2-simple.stp* Sample Output

4.2.7. Periodically Print I/O Block Time

This section describes how to track the amount of time each block I/O requests spends waiting for completion. This is useful in determining whether there are too many outstanding block I/O operations at any given time.

ioblktime.stp

```

#!/usr/bin/env stap

global req_time, etimes

probe ioblock.request
{
    req_time[$bio] = gettimeofday_us()
}

probe ioblock.end
{
    t = gettimeofday_us()
    s = req_time[$bio]
    delete req_time[$bio]
    if (s) {
        etimes[devname, bio_rw_str(rw)] <<< t - s
    }
}

probe timer.s(10), end {
    ansi_clear_screen()
    printf("%10s %3s %10s %10s %10s\n",
           "device", "rw", "total (us)", "count", "avg (us)")
    foreach ([dev,rw] in etimes - limit 20) {
        printf("%10s %3s %10d %10d %10d\n", dev, rw,
               @sum(etimes[dev,rw]), @count(etimes[dev,rw]), @avg(etimes[dev,rw]))
    }
    delete etimes
}

```

ioblktime.stp computes the average waiting time for block I/O per device, and prints a list every 10 seconds. As always, you can revise this refresh rate by editing the specified value in **probe timer.s(10), end {**.

In some cases, there can be too many outstanding block I/O operations, at which point the script can exceed the default number of **MAXMAPENTRIES**. **MAXMAPENTRIES** is the maximum number of rows in an array if the array size is not specified explicitly when declared. If the script exceeds the default **MAXMAPENTRIES** value of 2048, run the script again with the **stap** option - **DMAXMAPENTRIES=10000**.

device	rw	total (us)	count	avg (us)
sda	W	9659	6	1609
dm-0	W	20278	6	3379
dm-0	R	20524	5	4104
sda	R	19277	5	3855

Example 4.12. *ioblktime.stp* Sample Output

Example 4.12, “*ioblktime.stp* Sample Output” displays the device name, operations performed (**rw**), total wait time of all operations (**total(us)**), number of operations (**count**), and average wait time for all those operations (**avg (us)**). The times tallied by the script are in microseconds.

4.3. Profiling

The following sections showcase scripts that profile kernel activity by monitoring function calls.

4.3.1. Counting Function Calls Made

This section describes how to identify how many times the system called a specific kernel function in a 30-second sample. Depending on your use of wildcards, you can also use this script to target multiple kernel functions.

`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()
}
```

`functioncallcount.stp` takes the targeted kernel function as an argument. The argument supports wildcards, which enables you to target multiple kernel functions up to a certain extent.

You can increase the sample time by editing the timer in the second probe (`timer.ms()`). The output of `functioncallcount.stp` contains the name of the function called and how many times it was called during the sample time (in alphabetical order). [Example 4.13, “functioncallcount.stp Sample Output”](#) contains an excerpt from the output of `stap countcalls.stp "**@mm/*.c"`:

```
[...]
__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
[...]
```

Example 4.13. *functioncallcount.stp* Sample Output

4.3.2. Call Graph Tracing

This section describes how to trace incoming and outgoing function calls.

para-callgraph-simple.stp

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

global trace
probe kernel.function(@1).call {
    if (execname() == "stapio") next # skip our own helper process
    trace[tid()] = 1
    trace(1)
}
probe kernel.function(@1).return {
    trace(-1)
    delete trace[tid()]
}

probe kernel.function(@2).call { trace(1) }
probe kernel.function(@2).return { trace(-1) }
function trace(entry_p) {
    if(tid() in trace)
        printf("%s%s\n", thread_indent(entry_p),
            (entry_p>0?"->":"<-"),
            probefunc())
}

global trace
probe kernel.function(@1).call {
    if (execname() == "stapio") next # skip our own helper process
    trace[tid()] = 1
    trace(1)
}
probe kernel.function(@1).return {
    trace(-1)
    delete trace[tid()]
}

probe kernel.function(@2).call { trace(1) }
probe kernel.function(@2).return { trace(-1) }
```

para-callgraph-simple.stp takes two command-line arguments:

- A *trigger function* (**@1**), which enables or disables tracing on a per-thread basis. Tracing in each thread will continue as long as the trigger function has not exited yet.
- The kernel function/s whose entry/exit call you'd like to trace (**@2**).

[*para-callgraph-simple.stp*](#) uses **thread_indent()**; as such, its output contains the timestamp, process name, and thread ID of **@2** (i.e. the probe function you are tracing). For more information about **thread_indent()**, refer to its entry in [SystemTap Functions](#).

The following example contains an excerpt from the output for **stap para-callgraph.stp sys_read '@fs/*.c'**:

```
[...]
 0 klogd(1391):->sys_read
14 klogd(1391): ->fget_light
22 klogd(1391): <-fget_light
27 klogd(1391): ->vfs_read
35 klogd(1391): ->rw_verify_area
43 klogd(1391): <-rw_verify_area
49 klogd(1391): ->kmsg_read
 0 sendmail(1696):->sys_read
17 sendmail(1696): ->fget_light
26 sendmail(1696): <-fget_light
34 sendmail(1696): ->vfs_read
44 sendmail(1696): ->rw_verify_area
52 sendmail(1696): <-rw_verify_area
58 sendmail(1696): ->proc_file_read
70 sendmail(1696): ->loadavg_read_proc
84 sendmail(1696): ->proc_calc_metrics
92 sendmail(1696): <-proc_calc_metrics
95 sendmail(1696): <-loadavg_read_proc
101 sendmail(1696): <-proc_file_read
106 sendmail(1696): ->dnotify_parent
115 sendmail(1696): <-dnotify_parent
119 sendmail(1696): ->inotify_dentry_parent_queue_event
127 sendmail(1696): <-inotify_dentry_parent_queue_event
133 sendmail(1696): ->inotify_inode_queue_event
141 sendmail(1696): <-inotify_inode_queue_event
146 sendmail(1696): <-vfs_read
151 sendmail(1696):<-sys_read
```

Example 4.14. *para-callgraph-simple.stp* Sample Output

4.3.3. Determining Time Spent in Kernel and User Space

This section illustrates how to determine the amount of time any given thread is spending in either kernel or user-space.

thread-times.stp

```
#!/usr/bin/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
    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
}

```

[thread-times.stp](#) lists the top 20 processes currently taking up CPU time within a 5-second sample, along with the total number of CPU ticks made during the sample. The output of this script also notes the percentage of CPU time each process used, as well as whether that time was spent in kernel space or user space.

Example 4.15, “thread-times.stp Sample Output” contains a 5-second sample of the output for [thread-times.stp](#):

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%

Example 4.15. thread-times.stp Sample Output

4.3.4. Monitoring Polling Applications

This section how to identify and monitor which applications are polling. Doing so allows you to track unnecessary or excessive polling, which can help you pinpoint areas for improvement in terms of CPU usage and power savings.

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])
  }
}
```

[timeout.stp](#) tracks how many times each application used the following system calls over time:

- **poll**
- **select**
- **epoll**
- **itimer**
- **futex**

- **nanosleep**
- **signal**

In some applications, these system calls are used excessively. As such, they are normally identified as "likely culprits" for polling applications. Note, however, that an application may be using a different system call to poll excessively; sometimes, it is useful to find out the top system calls used by the system (refer to [Section 4.3.5, "Tracking Most Frequently Used System Calls"](#) for instructions). Doing so can help you identify any additional suspects, which you can add to [timeout.stp](#) for tracking.

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	dhcdd
4189	6916	0	0	2	0	0	0	scim-panel-gtk
1863	5767	0	0	0	0	0	0	iscsid
2562	0	2881	0	1	0	1438	0	pcscd
4257	4255	0	0	1	0	0	0	gnome-power-man
4278	3876	0	0	60	0	0	0	multiloader-apple
4083	0	1331	0	1728	0	0	0	Xorg
3921	1603	0	0	0	0	0	0	gam_server
4248	1591	0	0	0	0	0	0	nm-applet
3165	0	1441	0	0	0	0	0	xterm
29548	0	1440	0	0	0	0	0	httpd
1862	0	0	0	0	0	1438	0	iscsid

Example 4.16. *timeout.stp* Sample Output

You can increase the sample time by editing the timer in the second probe (**timer.s()**). The output of [functioncallcount.stp](#) contains the name and UID of the top 20 polling applications, along with how many times each application performed each polling system call (over time). [Example 4.16, "timeout.stp Sample Output"](#) contains an excerpt of the script:

4.3.5. Tracking Most Frequently Used System Calls

[timeout.stp](#) from [Section 4.3.4, "Monitoring Polling Applications"](#) helps you identify which applications are polling by pointing out which ones used the following system calls most frequently:

- **poll**
- **select**
- **epoll**
- **itimer**
- **futex**
- **nanosleep**

- **signal**

However, in some systems, a different system call might be responsible for excessive polling. If you suspect that a polling application might be using a different system call to poll, you need to identify first the top system calls used by the system. To do this, use [topsys.stp](#).

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")
}
```

[topsys.stp](#) lists the top 20 system calls used by the system per 5-second interval. It also lists how many times each system call was used during that period. Refer to [Example 4.17, “topsys.stp Sample Output”](#) for a sample output.

```
-----
      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
-----

```

Example 4.17. *topsys.stp* Sample Output

4.3.6. Tracking System Call Volume Per Process

This section illustrates how to determine which processes are performing the highest volume of system calls. In previous sections, we've described how to monitor the top system calls used by the system over time ([Section 4.3.5, "Tracking Most Frequently Used System Calls"](#)). We've also described how to identify which applications use a specific set of "polling suspect" system calls the most ([Section 4.3.4, "Monitoring Polling Applications"](#)). Monitoring the volume of system calls made by each process provides more data in investigating your system for polling processes and other resource hogs.

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)
}

```

syscalls_by_proc.stp lists the top 20 processes performing the highest number of system calls. It also lists how many system calls each process performed during the time period. Refer to [Example 4.18, "topsys.stp Sample Output"](#) for a sample output.

```

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      dhcdd
84      scim-bridge
78      gnome-screensav
66      scim-launcher
[...]
```

Example 4.18. *topsys.stp* Sample Output

If you prefer the output to display the process IDs instead of the process names, use the following script instead.

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)
}
```

As indicated in the output, you need to manually exit the script in order to display the results. You can add a timed expiration to either script by simply adding a **timer.s()** probe; for example, to instruct the script to expire after 5 seconds, add the following probe to the script:

```
probe timer.s(5)
{
```

```
exit()
}
```

4.4. Identifying Contended User-Space Locks

This section describes how to identify contended user-space locks throughout the system within a specific time period. The ability to identify contended user-space locks can help you investigate hangs that you suspect may be caused by **futex** contentions.

Simply put, a **futex** contention occurs when multiple processes are trying to access the same region of memory. In some cases, this can result in a deadlock between the processes in contention, thereby appearing as an application hang.

To do this, [futexes.stp](#) probes the **futex** system call.

[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]))
}
```

[futexes.stp](#) needs to be manually stopped; upon exit, it prints the following information:

- Name and ID of the process responsible for a contention
- The region of memory it contested
- How many times the region of memory was contended
- Average time of contention throughout the probe

Example 4.19, “[futexes.stp Sample Output](#)” contains an excerpt from the output of [futexes.stp](#) upon exiting the script (after approximately 20 seconds).

```
[...]  
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  
[...]
```

Example 4.19. [futexes.stp](#) Sample Output

Understanding SystemTap Errors

This chapter explains the most common errors you may encounter while using SystemTap.

5.1. Parse and Semantic Errors

These types of errors occur while SystemTap attempts to parse and translate the script into C, prior to being converted into a kernel module. For example type errors result from operations that assign invalid values to variables or arrays.

parse error: expected *foo*, saw *bar*

The script contains a grammatical/typographical error. SystemTap detected type of construct that is incorrect, given the context of the probe.

The following invalid SystemTap script is missing its probe handlers:

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

It results in the following error message showing that the parser was expecting something other than the **probe** keyword in column 1 of line 2:

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

parse error: embedded code in unprivileged script

The script contains unsafe embedded C code (blocks of code surrounded by `%{ %}`). SystemTap allows you to embed C code in a script, which is useful if there are no tapsets to suit your purposes. However, embedded C constructs are not be safe; as such, SystemTap warns you with this error if such constructs appear in the script.

If you are sure of the safety of any similar constructs in the script *and* are member of **stapdev** group (or have root privileges), run the script in "guru" mode by using the option **-g** (i.e. **stap -g script**).

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

The function **foo** in the script used the wrong type (i.e. **%s** or **%d**). This error will present itself in [Example 5.1, "error-variable.stp"](#), because the function **execname()** returns a string the format specifier should be a **%s**, not **%d**.

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

[Example 5.1. error-variable.stp](#)

semantic error: unresolved type for identifier '*foo*'

The identifier (e.g. a variable) was used, but no type (integer or string) could be determined. This occurs, for instance, if you use a variable in a **printf** statement while the script never assigns a value to the variable.

semantic error: Expecting symbol or array index expression

SystemTap could not assign a value to a variable or to a location in an array. The destination for the assignment is not a valid destination. The following example code would generate this error:

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

while searching for arity *N* function, semantic error: unresolved function call

A function call or array index expression in the script used an invalid number of arguments/parameters. In SystemTap *arity* can either refer to the number of indices for an array, or the number of parameters to a function.

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

The script used an array operation without declaring the array as a global variable (global variables can be declared after their use in Systemtap scripts). Similar messages appear if an array is used, but with inconsistent arities.

semantic error: variable '*foo*' modified during '*foreach*' iteration

The array **foo** is being modified (being assigned to or deleted from) within an active **foreach** loop. This error also displays if an operation within the script performs a function call within the **foreach** loop.

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

SystemTap did not understand what the event or SystemTap function **foo** refers to. This usually means that SystemTap could not find a match for **foo** in the tapset library. The *N* refers to the line and column of the error.

semantic error: no match for probe point, while resolving probe point *foo*

The events / handler function **foo** could not be resolved altogether, for a variety of reasons. This error occurs when the script contains the event **kernel.function("blah")**, and **blah** does not exist. In some cases, the error could also mean the script contains an invalid kernel file name or source line number.

semantic error: unresolved target-symbol expression

A handler in the script references a target variable, but the value of the variable could not be resolved. This error could also mean that a handler is referencing a target variable that is not valid in the context when it was referenced. This may be a result of compiler optimization of the generated code.

semantic error: libdwfl failure

There was a problem processing the debugging information. In most cases, this error results from the installation of a **kernel-debuginfo** RPM whose version does not match the probed kernel exactly. The installed **kernel-debuginfo** RPM itself may have some consistency / correctness problems.

semantic error: cannot find *foo* debuginfo

SystemTap could not find a suitable **kernel-debuginfo** at all.

5.2. Run Time Errors and Warnings

Runtime errors and warnings occur when the SystemTap instrumentation has been installed and is collecting data on the system.

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

Errors and/or skipped probes occurred during this run. Both *N* and *M* are the counts of the number of probes that were not executed due to conditions such as too much time required to execute event handlers over an interval of time.

division by 0

The script code performed an invalid division.

aggregate element not found

An statistics extractor function other than **@count** was invoked on an aggregate that has not had any values accumulated yet. This is similar to a division by zero.

aggregation overflow

An array containing aggregate values contains too many distinct key pairs at this time.

MAXNESTING exceeded

Too many levels of function call nesting were attempted. The default nesting of function calls allowed is 10.

MAXACTION exceeded

The probe handler attempted to execute too many statements in the probe handler. The default number of actions allow in a probe handler is 1000.

kernel/user string copy fault at *ADDR*

The probe handler attempted to copy a string from kernel or user space at an invalid address (*ADDR*).

pointer dereference fault

There was a fault encountered during a pointer dereference operation such as a target variable evaluation.

References

This chapter enumerates other references for more information about SystemTap. It is advisable that you refer to these sources in the course of writing advanced probes and tapsets.

SystemTap Wiki

The *SystemTap Wiki* is a collection of links and articles related to the deployment, usage, and development of SystemTap. You can find it in <http://sourceware.org/systemtap/wiki/HomePage>.

SystemTap Tutorial

Much of the content in this book comes from the *SystemTap Tutorial*. The *SystemTap Tutorial* is a more appropriate reference for users with intermediate to advanced knowledge of C++ and kernel development, and can be found at <http://sourceware.org/systemtap/tutorial/>.

man stapprobes

The **stapprobes** man page enumerates a variety of probe points supported by SystemTap, along with additional aliases defined by the SystemTap tapset library. The bottom of the man page includes a list of other man pages enumerating similar probe points for specific system components, such as **stapprobes.scsi**, **stapprobes.kprocess**, **stapprobes.signal**, etc.

man stapfuncs

The **stapfuncs** man page enumerates numerous functions supported by the SystemTap tapset library, along with the prescribed syntax for each one. Note, however, that this is not a complete list of *all* supported functions; there are more undocumented functions available.

SystemTap Language Reference

This document is a comprehensive reference of SystemTap's language constructs and syntax. It is recommended for users with a rudimentary to intermediate knowledge of C++ and other similar programming languages. The *SystemTap Language Reference* is available to all users at <http://sourceware.org/systemtap/langref/>

Tapset Developers Guide

Once you have sufficient proficiency in writing SystemTap scripts, you can then try your hand out on writing your own tapsets. The *Tapset Developers Guide* describes how to add functions to your tapset library.

Test Suite

The **systemtap-testsuite** package allows you to test the entire SystemTap toolchain without having to build from source. In addition, it also contains numerous examples of SystemTap scripts you can study and test; some of these scripts are also documented in [Chapter 4, Useful SystemTap Scripts](#).

By default, the example scripts included in **systemtap-testsuite** are located in **/usr/share/systemtap/testsuite/systemtap.examples**.

Appendix A. Revision History

Revision 2.0 Mon Jul 20 2009 Don Domingo ddomingo@redhat.com
includes 5.4 minor updates and additional script "dropwatch.stp"

Revision 1.0 Wed Jun 17 2009 Don Domingo ddomingo@redhat.com
Building+pushing to RHEL

Index

Symbols

\$count

- sample usage

- local variables, 39

\$return

- sample usage

- local variables, 37, 40

@avg (integer extractor)

- computing for statistical aggregates

- array operations, 27

@count (integer extractor)

- computing for statistical aggregates

- array operations, 27

@max (integer extractor)

- computing for statistical aggregates

- array operations, 27

@min (integer extractor)

- computing for statistical aggregates

- array operations, 27

@sum (integer extractor)

- computing for statistical aggregates

- array operations, 27

A

adding values to statistical aggregates

- computing for statistical aggregates

- array operations, 27

advantages of cross-instrumentation, 6

aggregate element not found

- runtime errors/warnings

- understanding SystemTap errors, 59

aggregates (statistical)

- array operations, 26

aggregation overflow

- runtime errors/warnings

- understanding SystemTap errors, 59

algebraic formulas using arrays

- reading values from arrays

- array operations, 22

architecture notation, determining, 7

architecture of SystemTap, 11

array locals not supported

- parse/semantics error

- understanding SystemTap errors, 58

array operations

- assigning associated values, 22

- associating timestamps to process names, 22

associative arrays, 22

clearing arrays/array elements, 24

- delete operator, 24

- multiple array operations within the same probe, 25

- virtual file system reads (non-cumulative), tallying, 24

computing for statistical aggregates, 26

- @avg (integer extractor), 27

- @count (integer extractor), 27

- @max (integer extractor), 27

- @min (integer extractor), 27

- @sum (integer extractor), 27

- adding values to statistical aggregates, 27

- count (operator), 27

- extracting data collected by statistical aggregates, 27

conditional statements, using arrays in, 25

- testing for array membership, 26

deleting arrays and array elements, 24

incrementing associated values, 23

- tallying virtual file system reads (VFS reads), 23

multiple elements in an array, 23

processing multiple elements in an array, 23

- cumulative virtual file system reads, tallying, 23

- foreach, 23

- iterations, processing elements in an array

- as, 23

- limiting the output of foreach, 24

- ordering the output of foreach, 24

reading values from arrays, 22

- computing for timestamp deltas, 22

- empty unique keys, 23

- using arrays in simple computations, 22

arrays, 21

- (see also associative arrays)

assigning associated values

- array operations, 22

- associating timestamps to process names, 22

- associating timestamps to process names

- array operations, 22

associated values

- introduction

- arrays, 21

associating timestamps to process names

- assigning associated values

- array operations, 22

associative arrays

- introduction, 21
 - associated values, 21
 - example, 21
 - index expression, 21
 - key pairs, 21
 - syntax, 21
 - unique keys, 21
- asynchronous events
 - Events, 14

B

- begin
 - Events, 14
- building instrumentation modules from SystemTap scripts, 5
- building kernel modules from SystemTap scripts, 5

C

- call graph tracing
 - examples of SystemTap scripts, 46
- capabilities of SystemTap
 - Introduction, 1
- changes to file attributes, monitoring
 - examples of SystemTap scripts, 43
- clearing arrays/array elements
 - array operations, 24
 - delete operator, 24
 - multiple array operations within the same probe, 25
 - virtual file system reads (non-cumulative), tallying, 24
- command-line arguments
 - SystemTap handler constructs
 - handlers, 21
- compiling instrumentation/kernel modules from SystemTap scripts, 5
- components
 - SystemTap scripts
 - introduction, 11
- computing for statistical aggregates
 - array operations, 26
 - @avg (integer extractor), 27
 - @count (integer extractor), 27
 - @max (integer extractor), 27
 - @min (integer extractor), 27
 - @sum (integer extractor), 27
 - adding values to statistical aggregates, 27
 - count (operator), 27
 - extracting data collected by statistical aggregates, 27

- computing for timestamp deltas
 - reading values from arrays
 - array operations, 22
- conditional operators
 - conditional statements
 - handlers, 20
- conditional statements, using arrays in
 - array operations, 25
 - testing for array membership, 26
- CONFIG_HZ, computing for, 19
- contended user-space locks (futex contentions), identifying
 - examples of SystemTap scripts, 55
- copy fault
 - runtime errors/warnings
 - understanding SystemTap errors, 59
- count operator
 - computing for statistical aggregates
 - array (operator), 27
- counting function calls
 - examples of SystemTap scripts, 45
- CPU ticks
 - examples of SystemTap scripts, 48
- cpu()
 - functions, 16
- cross-compiling, 5
- cross-instrumentation
 - advantages of, 6
 - building kernel modules from SystemTap scripts, 5
 - configuration
 - host system and target system, 6
 - generating instrumentation from SystemTap scripts, 5
 - host system, 6
 - instrumentation module, 6
 - target kernel, 6
 - target system, 6
- ctime()
 - functions, 16
- ctime(), example of usage
 - script examples, 37
- cumulative I/O, tracking
 - examples of SystemTap scripts, 39
- cumulative virtual file system reads, tallying
 - processing multiple elements in an array
 - array operations, 23

D

- delete operator
 - clearing arrays/array elements

- array operations, 24
- determining architecture notation, 7
- determining the kernel version, 3
- determining time spent in kernel and user space
 - examples of SystemTap scripts, 47
- device I/O, monitoring
 - examples of SystemTap scripts, 41
- device number of a file (integer format)
 - examples of SystemTap scripts, 42
- disk I/O traffic, summarizing
 - script examples, 35
- division by 0
 - runtime errors/warnings
 - understanding SystemTap errors, 59
- documentation goals
 - Introduction, 1

E

- embedded code in unprivileged script
 - parse/semantics error
 - understanding SystemTap errors, 57
- empty unique keys
 - reading values from arrays
 - array operations, 23
- end
- Events, 14
- errors
 - parse/semantics error, 57
 - embedded code in unprivileged script, 57
 - expected symbol/array index expression, 58
 - grammatical/typographical script error, 57
 - guru mode, 57
 - invalid values to variables/arrays, 57
 - libdwfl failure, 59
 - no match for probe point, 58
 - non-global arrays, 58
 - probe mismatch, 58
 - type mismatch for identifier, 57
 - unresolved function call, 58
 - unresolved target-symbol expression, 58
 - unresolved type for identifier, 58
 - variable modified during 'foreach', 58
 - runtime errors/warnings, 59
 - aggregate element not found, 59
 - aggregation overflow, 59
 - copy fault, 59
 - division by 0, 59
 - MAXACTION exceeded, 59
 - MAXNESTING exceeded, 59
 - number of errors: N, skipped probes: M, 59
 - pointer dereference fault, 59

- event types
 - Understanding How SystemTap Works, 11
- Events
 - asynchronous events, 14
 - begin, 14
 - end, 14
 - examples of synchronous and asynchronous events, 13
 - introduction, 13
 - kernel.function("function"), 13
 - kernel.trace("tracepoint"), 13
 - module("module"), 13
 - synchronous events, 13
 - syscall.system_call, 13
 - timer events, 14
 - vfs.file_operation, 13
 - wildcards, 13
- events and handlers, 11
- events wildcards, 13
- example
 - introduction
 - arrays, 21
- example of multiple command-line arguments
 - examples of SystemTap scripts, 46
- examples of synchronous and asynchronous events
 - Events, 13
- examples of SystemTap scripts, 29
 - call graph tracing, 46
 - CPU ticks, 48
 - ctime(), example of usage, 37
 - determining time spent in kernel and user space, 47
 - file device number (integer format), 42
 - futex system call, 55
 - identifying contended user-space locks (i.e. futex contentions), 55
 - if/else conditionals, alternative syntax, 30
 - inode number, 42
 - monitoring changes to file attributes, 43
 - monitoring device I/O, 41
 - monitoring I/O block time, 43
 - monitoring I/O time, 37
 - monitoring incoming TCP connections, 32
 - monitoring polling applications, 49
 - monitoring reads and writes to a file, 42
 - monitoring system calls, 51
 - monitoring system calls (volume per process), 53
 - monitoring TCP packets, 32

- multiple command-line arguments, example of, 46
- net/socket.c, tracing functions from, 31
- network profiling, 29, 34
- process deadlocks (arising from futex contentions), 55
- stat -c, determining file device number (integer format), 42
- stat -c, determining whole device number, 41
- summarizing disk I/O traffic, 35
- tallying function calls, 45
- thread_indent(), sample usage, 47
- timer.ms(), sample usage, 45
- timer.s(), sample usage, 51, 52
- tracing functions called in network socket code, 31
- tracking cumulative I/O, 39
- trigger function, 47
- usrdev2kerndev(), 41
- whole device number (usage as a command-line argument), 41
- exceeded MAXACTION
 - runtime errors/warnings
 - understanding SystemTap errors, 59
- exceeded MAXNESTING
 - runtime errors/warnings
 - understanding SystemTap errors, 59
- exit()
 - functions, 15
- expected symbol/array index expression
 - parse/semantics error
 - understanding SystemTap errors, 58
- extracting data collected by statistical aggregates
 - computing for statistical aggregates
 - array operations, 27

F

- fedoradebugurl.sh, 5
- feedback
 - contact information for this manual, vii
- file attributes, monitoring changes to
 - examples of SystemTap scripts, 43
- file device number (integer format)
 - examples of SystemTap scripts, 42
- file reads/writes, monitoring
 - examples of SystemTap scripts, 42
- for loops
 - conditional statements
 - handlers, 20
- foreach
 - processing multiple elements in an array

- array operations, 23
- format
 - introduction
 - arrays, 21
- format and syntax
 - printf(), 15
 - SystemTap handler constructs
 - handlers, 18
 - SystemTap scripts
 - introduction, 12
- format specifiers
 - printf(), 15
- format strings
 - printf(), 15
- function call (unresolved)
 - parse/semantics error
 - understanding SystemTap errors, 58
- function calls (incoming/outgoing), tracing
 - examples of SystemTap scripts, 46
- function calls, tallying
 - examples of SystemTap scripts, 45
- functions, 16
 - cpu(), 16
 - ctime(), 16
 - gettimeofday_s(), 16
 - pp(), 16
 - SystemTap scripts
 - introduction, 12
 - target(), 18
 - thread_indent(), 16
 - tid(), 16, 16
 - uid(), 16
- functions (used in handlers)
 - exit(), 15
- functions called in network socket code, tracing
 - examples of SystemTap scripts, 31
- futex contention, definition
 - examples of SystemTap scripts, 55
- futex contentions, identifying
 - examples of SystemTap scripts, 55
- futex system call
 - examples of SystemTap scripts, 55

G

- gettimeofday_s()
 - functions, 16
- global
 - SystemTap handler constructs
 - handlers, 18
- goals, documentation
 - Introduction, 1

- grammatical/typographical script error
 - parse/semantics error
 - understanding SystemTap errors, 57
- guru mode
 - parse/semantics error
 - understanding SystemTap errors, 57

H

- handler functions, 16
- handlers
 - conditional statements, 19
 - conditional operators, 20
 - for loops, 20
 - if/else, 19
 - while loops, 20
- introduction, 15
- SystemTap handler constructs, 18
 - command-line arguments, 21
 - global, 18
 - syntax and format, 18
 - variable notations, 21
 - variables, 18
- handlers and events, 11
 - SystemTap scripts
 - introduction, 11
- heaviest disk reads/writes, identifying
 - script examples, 35
- host system
 - cross-instrumentation, 6
- host system and target system
 - cross-instrumentation
 - configuration, 6

I

- I/O block time, monitoring
 - examples of SystemTap scripts, 43
- I/O monitoring (by device)
 - examples of SystemTap scripts, 41
- I/O time, monitoring
 - examples of SystemTap scripts, 37
- I/O traffic, summarizing
 - script examples, 35
- identifier type mismatch
 - parse/semantics error
 - understanding SystemTap errors, 57
- identifying contended user-space locks (i.e. futex contentions)
 - examples of SystemTap scripts, 55
- identifying heaviest disk reads/writes
 - script examples, 35
- if/else

- conditional statements
 - handlers, 19
- if/else conditionals, alternative syntax
 - examples of SystemTap scripts, 30
- if/else statements, using arrays in
 - array operations, 25
- incoming TCP connections, monitoring
 - examples of SystemTap scripts, 32
- incoming/outgoing function calls, tracing
 - examples of SystemTap scripts, 46
- incrementing associated values
 - array operations, 23
 - tallying virtual file system reads (VFS reads), 23
- index expression
 - introduction
 - arrays, 21
- initial testing, 5
- inode number
 - examples of SystemTap scripts, 42
- Installation
 - fedoradebugurl.sh, 5
 - initial testing, 5
 - kernel information packages, 3
 - kernel version, determining the, 3
 - required packages, 3
 - Setup and Installation, 3
 - systemtap package, 3
 - systemtap-runtime package, 3
- instrumentation module
 - cross-instrumentation, 6
- instrumentation modules from SystemTap scripts, building, 5
- integer extractors
 - computing for statistical aggregates
 - array operations, 27
- Introduction
 - capabilities of SystemTap, 1
 - documentation goals, 1
 - goals, documentation, 1
 - limitations of SystemTap, 2
 - performance monitoring, 1
- invalid division
 - runtime errors/warnings
 - understanding SystemTap errors, 59
- invalid values to variables/arrays
 - parse/semantics error
 - understanding SystemTap errors, 57
- iterations, processing elements in an array as
 - processing multiple elements in an array
 - array operations, 23

K

- kernel and user space, determining time spent in
 - examples of SystemTap scripts, 47
- kernel information packages, 3
- kernel modules from SystemTap scripts, building, 5
- kernel version, determining the, 3
- kernel.function("function")
 - Events, 13
- kernel.trace("tracepoint")
 - Events, 13
- key pairs
 - introduction
 - arrays, 21

L

- libdwfl failure
 - parse/semantics error
 - understanding SystemTap errors, 59
- limitations of SystemTap
 - Introduction, 2
- limiting the output of foreach
 - processing multiple elements in an array
 - array operations, 24
- local variables
 - name, 18
 - sample usage
 - \$count, 39
 - \$return, 37, 40

M

- MAXACTION exceeded
 - runtime errors/warnings
 - understanding SystemTap errors, 59
- MAXNESTING exceeded
 - runtime errors/warnings
 - understanding SystemTap errors, 59
- membership (in array), testing for
 - conditional statements, using arrays in
 - array operations, 26
- module("module")
 - Events, 13
- monitoring changes to file attributes
 - examples of SystemTap scripts, 43
- monitoring cumulative I/O
 - examples of SystemTap scripts, 39
- monitoring device I/O
 - examples of SystemTap scripts, 41
- monitoring I/O block time
 - examples of SystemTap scripts, 43
- monitoring I/O time

- examples of SystemTap scripts, 37
- monitoring incoming TCP connections
 - examples of SystemTap scripts, 32
- monitoring polling applications
 - examples of SystemTap scripts, 49
- monitoring reads and writes to a file
 - examples of SystemTap scripts, 42
- monitoring system calls
 - examples of SystemTap scripts, 51
- monitoring system calls (volume per process)
 - examples of SystemTap scripts, 53
- monitoring TCP packets
 - examples of SystemTap scripts, 32
- multiple array operations within the same probe
 - clearing arrays/array elements
 - array operations, 25
- multiple command-line arguments, example of
 - examples of SystemTap scripts, 46
- multiple elements in an array
 - array operations, 23

N

- name
 - local variables, 18
- net/socket.c, tracing functions from
 - examples of SystemTap scripts, 31
- network profiling
 - examples of SystemTap scripts, 29, 34
- network socket code, tracing functions called in
 - examples of SystemTap scripts, 31
- network traffic, monitoring
 - examples of SystemTap scripts, 29, 34
- no match for probe point
 - parse/semantics error
 - understanding SystemTap errors, 58
- non-global arrays
 - parse/semantics error
 - understanding SystemTap errors, 58
- number of errors: N, skipped probes: M
 - runtime errors/warnings
 - understanding SystemTap errors, 59

O

- operations
 - assigning associated values
 - associating timestamps to process names, 22
 - associative arrays, 22
 - clearing arrays/array elements, 24
 - delete operator, 24

-
- multiple array operations within the same probe, 25
 - virtual file system reads (non-cumulative), tallying, 24
 - computing for statistical aggregates, 26
 - @avg (integer extractor), 27
 - @count (integer extractor), 27
 - @max (integer extractor), 27
 - @min (integer extractor), 27
 - @sum (integer extractor), 27
 - adding values to statistical aggregates, 27
 - count (operator), 27
 - extracting data collected by statistical aggregates, 27
 - conditional statements, using arrays in, 25
 - testing for array membership, 26
 - deleting arrays and array elements, 24
 - incrementing associated values, 23
 - tallying virtual file system reads (VFS reads), 23
 - multiple elements in an array, 23
 - processing multiple elements in an array, 23
 - cumulative virtual file system reads, tallying, 23
 - foreach, 23
 - iterations, processing elements in an array as, 23
 - limiting the output of foreach, 24
 - ordering the output of foreach, 24
 - reading values from arrays, 22
 - computing for timestamp deltas, 22
 - empty unique keys, 23
 - using arrays in simple computations, 22
 - options, stap
 - Usage, 8
 - ordering the output of foreach
 - processing multiple elements in an array
 - array operations, 24
 - overflow of aggregation
 - runtime errors/warnings
 - understanding SystemTap errors, 59
- P**
- packages required to run SystemTap, 3
 - parse/semantics error
 - understanding SystemTap errors, 57
 - embedded code in unprivileged script, 57
 - expected symbol/array index expression, 58
 - grammatical/typographical script error, 57
 - guru mode, 57
 - invalid values to variables/arrays, 57
 - libdwfl failure, 59
 - no match for probe point, 58
 - non-global arrays, 58
 - probe mismatch, 58
 - type mismatch for identifier, 57
 - unresolved function call, 58
 - unresolved target-symbol expression, 58
 - unresolved type for identifier, 58
 - variable modified during 'foreach', 58
 - performance monitoring
 - Introduction, 1
 - pointer dereference fault
 - runtime errors/warnings
 - understanding SystemTap errors, 59
 - polling applications, monitoring
 - examples of SystemTap scripts, 49
 - pp()
 - functions, 16
 - printf()
 - format specifiers, 15
 - format strings, 15, 15
 - syntax and format, 15
 - printing I/O activity (cumulative)
 - examples of SystemTap scripts, 39
 - printing I/O block time (periodically)
 - examples of SystemTap scripts, 43
 - probe mismatch
 - parse/semantics error
 - understanding SystemTap errors, 58
 - probe point (no match for)
 - parse/semantics error
 - understanding SystemTap errors, 58
 - probes
 - SystemTap scripts
 - introduction, 12
 - process deadlocks (arising from futex contentions)
 - examples of SystemTap scripts, 55
 - processing multiple elements in an array
 - array operations, 23
 - cumulative virtual file system reads, tallying
 - array operations, 23
 - foreach
 - array operations, 23
 - limiting the output of foreach
 - array operations, 24
 - ordering the output of foreach
 - array operations, 24
 - profiling the network
 - examples of SystemTap scripts, 29, 34
-

R

- reading values from arrays
 - array operations, 22
 - empty unique keys, 23
 - using arrays in simple computations, 22
- computing for timestamp deltas
 - array operations, 22
- reads/writes to a file, monitoring
 - examples of SystemTap scripts, 42
- required packages, 3
- RPMs required to run SystemTap, 3
- running scripts from standard input, 8
- running SystemTap scripts
 - Usage, 7
- runtime errors/warnings
 - understanding SystemTap errors, 59
 - aggregate element not found, 59
 - aggregation overflow, 59
 - copy fault, 59
 - division by 0, 59
 - MAXACTION exceeded, 59
 - MAXNESTING exceeded, 59
 - number of errors: N, skipped probes: M, 59
 - pointer dereference fault, 59

S

- script examples
 - call graph tracing, 46
 - CPU ticks, 48
 - ctime(), example of usage, 37
 - determining time spent in kernel and user space, 47
 - file device number (integer format), 42
 - futex system call, 55
 - identifying contended user-space locks (i.e. futex contentions), 55
 - if/else conditionals, alternative syntax, 30
 - inode number, 42
 - monitoring changes to file attributes, 43
 - monitoring device I/O, 41
 - monitoring I/O block time, 43
 - monitoring I/O time, 37
 - monitoring incoming TCP connections, 32
 - monitoring polling applications, 49
 - monitoring reads and writes to a file, 42
 - monitoring system calls, 51
 - monitoring system calls (volume per process), 53
 - monitoring TCP packets, 32
 - multiple command-line arguments, example of, 46

- net/socket.c, tracing functions from, 31
- network profiling, 29, 34
- process deadlocks (arising from futex contentions), 55
- stat -c, determining file device number (integer format), 42
- stat -c, determining whole device number, 41
- summarizing disk I/O traffic, 35
- tallying function calls, 45
- thread_indent(), sample usage, 47
- timer.ms(), sample usage, 45
- timer.s(), sample usage, 51, 52
- tracing functions called in network socket code, 31
- tracking cumulative I/O, 39
- trigger function, 47
- usrdev2kerndev(), 41
- whole device number (usage as a command-line argument), 41
- scripts
 - introduction, 11
 - components, 11
 - events and handlers, 11
 - format and syntax, 12
 - functions, 12
 - probes, 12
 - statement blocks, 12
- sessions, SystemTap, 11
- Setup and Installation, 3
- standard input, running scripts from
 - Usage, 8
- stap
 - Usage, 7
- stap options, 8
- stapdev
 - Usage, 7
- staprun
 - Usage, 7
- stapusr
 - Usage, 8
- stat -c, determining file device number (integer format)
 - examples of SystemTap scripts, 42
- stat -c, determining whole device number
 - examples of SystemTap scripts, 41
- statement blocks
 - SystemTap scripts
 - introduction, 12
- statistical aggregates
 - array operations, 26
- summarizing disk I/O traffic

- script examples, 35, 35
- synchronous events
 - Events, 13
- syntax
 - introduction
 - arrays, 21
- syntax and format
 - printf(), 15
 - SystemTap handler constructs
 - handlers, 18
 - SystemTap scripts
 - introduction, 12
- syscall.system_call
 - Events, 13
- system calls volume (per process), monitoring
 - examples of SystemTap scripts, 53
- system calls, monitoring
 - examples of SystemTap scripts, 51
- SystemTap architecture, 11
- SystemTap handlers
 - SystemTap handler constructs, 18
 - syntax and format, 18
- systemtap package, 3
- SystemTap script functions, 16
- SystemTap scripts
 - introduction, 11
 - components, 11
 - events and handlers, 11
 - format and syntax, 12
 - functions, 12
 - probes, 12
 - statement blocks, 12
 - useful examples, 29
- SystemTap scripts, how to run, 7
- SystemTap sessions, 11
- SystemTap statements
 - conditional statements, 19
 - conditional operators, 20
 - for loops, 20
 - if/else, 19
 - while loops, 20
 - SystemTap handler constructs
 - command-line arguments, 21
 - global, 18
 - variable notations, 21
 - variables, 18
- systemtap-runtime package, 3
- systemtap-testsuite package
 - sample scripts, 29

T

- tallying function calls
 - examples of SystemTap scripts, 45
- tallying virtual file system reads (VFS reads)
 - incrementing associated values
 - array operations, 23
- Tapsets
 - definition of, 28
- target kernel
 - cross-instrumentation, 6
- target system
 - cross-instrumentation, 6
- target system and host system
 - configuration, 6
- target()
 - functions, 18
- target-symbol expression, unresolved
 - parse/semantics error
 - understanding SystemTap errors, 58
- TCP connections (incoming), monitoring
 - examples of SystemTap scripts, 32
- TCP packets, monitoring
 - examples of SystemTap scripts, 32, 32
- testing for array membership
 - conditional statements, using arrays in
 - array operations, 26
- testing, initial, 5
- thread_indent()
 - functions, 16
- thread_indent(), sample usage
 - examples of SystemTap scripts, 47
- tid()
 - functions, 16
- time of I/O
 - examples of SystemTap scripts, 37
- time spent in kernel/user space, determining
 - examples of SystemTap scripts, 47
- timer events
 - Events, 14
- timer.ms(), sample usage
 - examples of SystemTap scripts, 45
- timer.s(), sample usage
 - examples of SystemTap scripts, 51, 52
- timestamp deltas, computing for
 - reading values from arrays
 - array operations, 22
- timestamps, association thereof to process names
 - assigning associated values
 - array operations, 22
- tracepoint, 13, 34

- tracing call graph
 - examples of SystemTap scripts, 46
- tracing functions called in network socket code
 - examples of SystemTap scripts, 31
- tracing incoming/outgoing function calls
 - examples of SystemTap scripts, 46
- tracking cumulative I/O
 - examples of SystemTap scripts, 39
- trigger function
 - examples of SystemTap scripts, 47
- type mismatch for identifier
 - parse/semantics error
 - understanding SystemTap errors, 57
- typographical script error
 - parse/semantics error
 - understanding SystemTap errors, 57

U

- uid()
 - functions, 16
- uname -m, 7
- uname -r, 3
- understanding SystemTap errors
 - runtime errors/warnings, 59
 - aggregate element not found, 59
 - aggregation overflow, 59
 - copy fault, 59
 - division by 0, 59
 - MAXACTION exceeded, 59
 - MAXNESTING exceeded, 59
 - number of errors: N, skipped probes: M, 59
 - pointer dereference fault, 59
- Understanding How SystemTap Works, 11
 - architecture, 11
 - event types, 11
 - events and handlers, 11
 - SystemTap sessions, 11
- understanding SystemTap errors
 - parse/semantics error, 57
 - embedded code in unprivileged script, 57
 - expected symbol/array index expression, 58
 - grammatical/typographical script error, 57
 - guru mode, 57
 - invalid values to variables/arrays, 57
 - libdwfl failure, 59
 - no match for probe point, 58
 - non-global arrays, 58
 - probe mismatch, 58
 - type mismatch for identifier, 57
 - unresolved function call, 58
 - unresolved target-symbol expression, 58

- unresolved type for identifier, 58
 - variable modified during 'foreach', 58
- unique keys
 - introduction
 - arrays, 21
- unprivileged script, embedded code in
 - parse/semantics error
 - understanding SystemTap errors, 57
- unresolved function call
 - parse/semantics error
 - understanding SystemTap errors, 58
- unresolved target-symbol expression
 - parse/semantics error
 - understanding SystemTap errors, 58
- unresolved type for identifier
 - parse/semantics error
 - understanding SystemTap errors, 58
- unsafe embedded code in unprivileged script
 - parse/semantics error
 - understanding SystemTap errors, 57
- Usage
 - options, stap, 8
 - running SystemTap scripts, 7
 - standard input, running scripts from, 8
 - stap, 7
 - stapdev, 7
 - staprun, 7
 - stapusr, 8
- useful examples of SystemTap scripts, 29
- user and kernel space, determining time spent in
 - examples of SystemTap scripts, 47
- using arrays in simple computations
 - reading values from arrays
 - array operations, 22
- Using SystemTap, 3
- usrdev2kerndev()
 - examples of SystemTap scripts, 41

V

- values, assignment of
 - array operations, 22
- variable modified during 'foreach'
 - parse/semantics error
 - understanding SystemTap errors, 58
- variable notations
 - SystemTap handler constructs
 - handlers, 21
- variables
 - SystemTap handler constructs
 - handlers, 18
- variables (local)

- name, 18
- sample usage
 - \$count, 39
 - \$return, 37, 40
- VFS reads, tallying of
 - incrementing associated values
 - array operations, 23
- vfs.file_operation
 - Events, 13
- virtual file system reads (cumulative), tallying
 - processing multiple elements in an array
 - array operations, 23
- virtual file system reads (non-cumulative), tallying
 - clearing arrays/array elements
 - array operations, 24

W

- while loops
 - conditional statements
 - handlers, 20
- whole device number (usage as a command-line argument)
 - examples of SystemTap scripts, 41
- wildcards in events, 13
- writes/reads to a file, monitoring
 - examples of SystemTap scripts, 42

