



Part—II

# SystemTap Tutorial

Last month, we explored SystemTap and how it works. We also discussed Tapset libraries, and looked at a few examples. Let's proceed from there.

I am sure you have gone through the examples given on the SystemTap website (<http://sourceware.org/systemtap/examples/index.html>) and looked at other resources I mentioned in the *References* section of the last article. I will not try to cover all the topics mentioned there.

To collect statistical data, SystemTap supports aggregate variables. These are important when data needs to be collected quickly and in large volumes. The “<<<” operator is used for aggregation. The left operand specifies a scalar or array-index l-value, which must be declared global. The right operand is a numeric expression.

Let's update the example (*stapcount.stp*) we saw last time, to use aggregation:

```
global syscalls
probe syscall.* { syscalls[name] <<< 1 }
probe timer.s(10) {
    foreach(name in syscalls- limit 5)
        printf("%8d %s\n", @count(syscalls[name]), name)
    delete syscalls
}
```

You might wonder that if it serves the same purpose, why should we use aggregation?

The collected data can later be extracted via different extractor functions like `@count(var)`,

`@min(var)`, `@max(var)`, etc. You can also generate nice-looking histograms with the `@hist_log` and `@hist_linear` functions. For more information, please refer to `/usr/share/doc/systemtap-<version>/langref.pdf`.

## Call tracing

Wouldn't it be good to know the flow of execution through the code, while debugging a problem?

SystemTap makes this very simple. Let's say you want to know all the “ext3”-related functions that are called when you create a file on an *ext3* file-system.

As my installed distribution uses the (default) *ext4* file-system on the root partition, I created a loop-back device, and formatted it as *ext3*. The call trace can provide a lot of information, which might be difficult to follow. But by having just one device formatted with *ext3*, I have narrowed down my focus. So, here is the script:

```
probe kernel.function("**@fs/ext3/**") {
    printf ("%s -> %s\n", thread_indent(1),
    probefunc())
}

probe kernel.function("**@fs/ext3/**").return {
    printf ("%s <- %s\n", thread_indent(-1), probefunc())
}
```

The `thread_indent` function provides a way to better organise the printed results. It 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.

After running the script, when you create a file (from another terminal window, on `ex3` formatted filesystem), you’ll get the following output:

```
0 touch(1546): -> ext3_permission
1792 touch(1546): <- ext3_permission
0 touch(1546): -> ext3_permission
91 touch(1546): <- ext3_permission
0 touch(1546): -> ext3_lookup
66 touch(1546): -> ext3_find_entry
123 touch(1546): -> ext3_getblk
171 touch(1546): -> ext3_get_blocks_handle
214 touch(1546): -> ext3_block_to_path
250 touch(1546): <- ext3_block_to_path
.....
.....
```

Similarly, to trace all functions in a program, run the following command:

```
$ stap -e 'probe process("<program_path>").function("**") { log(pp()) }' -c
<program_path>
```

You’ll need the `debuginfo` for the package that provides the program. For example, for `/bin/ls`, you’ll need the `debuginfo` of `coreutils`.

## Kernel statement

You can put a probe on a specific line number in the kernel code. With the following code, you can probe when line number 836 of `fs/open.c` is executed, and then print the local variables of the running function.

```
probe kernel.statement("**@fs/open.c:836") {
    printf("local variables %s \n", $$locals);
}
```

You’ll get an output similar to what’s shown below:

```
local variables inode=0xffff880008c6a7b0 error=0x0
```

## Guru mode

Guru mode allows you to run the script without any memory and data protection, which is normally provided by the SystemTap access restrictions. It allows you to change things, rather than just observing. To run the script in guru mode, you need to run it with the `-g` option. You can then modify any data in the function, at a memory address, etc. You can add

custom C code anywhere. However, note that you could easily mess something up and crash the kernel, so be very cautious when you use this mode.

## Embedded C functions

General syntax:

```
function <name>:<type> ( <arg1>:<type>, ... ) %{ <C_stmts> %}
```

Embedded C code is permitted in a function body. In that case, the script language body is replaced entirely by a piece of C code enclosed between `%{` and `%}` markers. We’ll pass the arguments to this function, and will get a return value. To do this communication SystemTap implements the `THIS` structure. The `THIS` structure is dynamic, and contains the arguments you specify as variables, and a field called `__retvalue`, which is the value that will be returned to the call site.

Let’s try to understand this, using an example from the Tapset library. To get the nice value of a process, just call `task_nice` from your `stap` script. Internally, in `/usr/share/systemtap/tapset/task.stp`, this function is defined as:

```
// Return the nice value of the given task
function task_nice:long (task:long) %{ /* pure */
    struct task_struct *t = (struct task_struct*)(long)THIS->task;
    THIS->__retvalue = kread(&(t->static_prio)) - MAX_RT_PRIO - 20;
    CATCH_DEREF_FAULT();
    %}
```

The function takes a long argument (`task`), and returns a long (nice value). Typecast the argument to `task_struct`, and later return the nice value by setting `THIS->__retvalue`.

## Fault injection

As you can probe any kernel function and modify kernel variables, you can use these features to inject errors. Let’s say you want to return an ‘Invalid argument’ error when a directory with a certain name is created.

From the kernel source:

```
SYSCALL_DEFINE2(mkdir, const char __user *, pathname, int, mode)
{
    return sys_mkdirat(AT_FDCWD, pathname, mode);
}
```

Now, let’s put a probe when `sys_mkdir` returns and modify the return value if the directory name matches “mydir”:

```
probe kernel.function("sys_mkdir").return {
    if (isinstr(user_string($pathname), "mydir")) {
        $return = -22;
    }
}
```

After running this script in guru mode, whenever you try to create a directory with the name “mydir”, you will get an ‘Invalid argument’ error.

## Tracing for custom modules

Now, let’s suppose you want to trace functions from custom modules that you have written. For instance: shown below are *module1* and *module2*. *Module1* exports a function which *module2* uses:

```
=====module1=====
#include <linux/module.h>

static int __init my_init( void )
{
    printk("Hello world from module 1 \n");
    return 0;
}

static void __exit my_exit( void )
{
    printk("Goodbye world from module 1 \n");
}

static void mod1fun(void)
{
    printk(" YAHOO ! I got into mod1fun \n");
}

EXPORT_SYMBOL(mod1fun);

module_init( my_init );
module_exit( my_exit );
MODULE_LICENSE("GPL");

=====module2=====
#include <linux/module.h>
extern void mod1fun(void);

static int __init my_init( void )
{
    printk("Hello world from module2\n");
    mod1fun();
    return 0;
}

static void __exit my_exit( void )
{
    printk("Goodbye world from module2\n");
}

module_init( my_init );
module_exit( my_exit );
MODULE_LICENSE("GPL");
```

After compiling both modules, I have inserted *module1*. Now, to probe whenever *mod1fun* is called, I run the following script:

```
probe module("module1").function("**") {
    printf("%s was called from somewhere \n", probefunc());
}
```


```
Pass 1: parsed user script and 59 library script(s) in 240usr/30sys/271real
ms.
semantic error: missing x86_64 kernel/module debuginfo under '/
lib/modules/<kernel version>/build' while resolving probe point
module("module1").function("**")
Pass 2: analyzed script: 0 probe(s), 0 function(s), 0 embed(s), 0 global(s)
in 90usr/380sys/474real ms.
Pass 2: analysis failed. Try again with another '--vp 01' option.
...
```

As the output shows, *stap* tries to search for information about *module1* under */lib/modules/<kernel version>/build*, but doesn’t find it, and gives an error. Let’s create a directory under */lib/modules/<kernel version>*, and copy *module1.ko* there:

```
$ mkdir /lib/modules/<kernel version> /custom
$ cp <Module_Path>/module1.ko /lib/modules/<kernel version> /custom/
```

Now, let’s run the *stap* script again. Then, as you load *module2*, you should see the following output in the terminal in which the script was running:

```
mod1fun was called from somewhere
```

In this manner, you can trace any custom modules; you just need to have *debuginfo* installed for those modules where *stap* can find it in Pass 2. The possibilities are endless. There are a lot of other interesting things you can do. I would recommend that you go through the references mentioned below, and try out as many examples as you can from the SystemTap website.  **END**

### References

<http://sourceware.org/systemtap/>  
<http://sourceware.org/systemtap/wiki/LW2008SystemTapTutorial>  
<http://sourceware.org/systemtap/wiki/HomePage?action=AttachFile&do=view&target=fosdem-stap.pdf>  
<http://www.redbooks.ibm.com/redpapers/pdfs/redp4469.p>

### By: Neependra Khare

The author is an open source enthusiast with a deep interest in Linux. He is currently working as a software engineer at KQ Infotech, and can be reached at neependra.khare@gmail.com. He also provides training on the Linux kernel and on debugging tools.