Using the Linux Kernel Tracepoints

Author: Mathieu Desnoyers

This document introduces Linux Kernel Tracepoints and their use. It provides examples of how to insert tracepoints in the kernel and connect probe functions to them and provides some examples of probe functions.

Purpose of tracepoints

A tracepoint placed in code provides a hook to call a function (probe) that you can provide at runtime. A tracepoint can be "on" (a probe is connected to it) or "off" (no probe is attached). When a tracepoint is "off" it has no effect, except for adding a tiny time penalty (checking a condition for a branch) and space penalty (adding a few bytes for the function call at the end of the instrumented function and adds a data structure in a separate section). When a tracepoint is "on", the function you provide is called each time the tracepoint is executed, in the execution context of the caller. When the function provided ends its execution, it returns to the caller (continuing from the tracepoint site).

You can put tracepoints at important locations in the code. They are lightweight hooks that can pass an arbitrary number of parameters, which prototypes are described in a tracepoint declaration placed in a header file.

They can be used for tracing and performance accounting.

Usage

Two elements are required for tracepoints :

- A tracepoint definition, placed in a header file.
- The tracepoint statement, in C code.

In order to use tracepoints, you should include linux/tracepoint.h.

In include/trace/events/subsys.h:

```
#undef TRACE_SYSTEM
#define TRACE_SYSTEM subsys

#if !defined(_TRACE_SUBSYS_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_SUBSYS_H

#include <linux/tracepoint.h>

DECLARE_TRACE(subsys_eventname,
        TP_PROTO(int firstarg, struct task_struct *p),
        TP_ARGS(firstarg, p));

#endif /* _TRACE_SUBSYS_H */

/* This part must be outside protection */
#include <trace/define_trace.h>
```

In subsys/file.c (where the tracing statement must be added):

```
#include <trace/events/subsys.h>

#define CREATE_TRACE_POINTS
DEFINE_TRACE(subsys_eventname);

void somefct(void)
{
        ...
        trace_subsys_eventname(arg, task);
        ...
}
```

Where :

- subsys_eventname is an identifier unique to your event
  - subsys is the name of your subsystem.
  - eventname is the name of the event to trace.
- *TP_PROTO(int firstarg, struct task_struct *p)* is the prototype of the function called by this tracepoint.
- *TP_ARGS(firstarg, p)* are the parameters names, same as found in the prototype.
- if you use the header in multiple source files, *#define CREATE_TRACE_POINTS* should appear only in one source file.

Connecting a function (probe) to a tracepoint is done by providing a probe (function to call) for the specific tracepoint through register_trace_subsys_eventname(). Removing a probe is done through unregister_trace_subsys_eventname(); it will remove the probe.

tracepoint_synchronize_unregister() must be called before the end of the module exit function to make sure there is no caller left using the probe. This, and the fact that preemption is disabled around the probe call, make sure that probe removal and module unload are safe.

The tracepoint mechanism supports inserting multiple instances of the same tracepoint, but a single definition must be made of a given tracepoint name over all the kernel to make sure no type conflict will occur. Name mangling of the tracepoints is done using the prototypes to make sure typing is correct. Verification of probe type correctness is done at the registration site by the compiler. Tracepoints can be put in inline functions, inlined static functions, and unrolled loops as well as regular functions.

The naming scheme "subsys_event" is suggested here as a convention intended to limit collisions. Tracepoint names are global to the kernel: they are considered as being the same whether they are in the core kernel image or in modules.

If the tracepoint has to be used in kernel modules, an EXPORT_TRACEPOINT_SYMBOL_GPL() or EXPORT_TRACEPOINT_SYMBOL() can be used to export the defined tracepoints.

If you need to do a bit of work for a tracepoint parameter, and that work is only used for the tracepoint, that work can be encapsulated within an if statement with the following:

```
if (trace_foo_bar_enabled()) {
        int i;
        int tot = 0;

        for (i = 0; i < count; i++)
                tot += calculate_nuggets();

        trace_foo_bar(tot);
}
```

All trace_<tracepoint>() calls have a matching trace_<tracepoint>_enabled() function defined that returns true if the tracepoint is enabled and false otherwise. The trace_<tracepoint>() should always be within the block of the if (trace_<tracepoint>_enabled()) to prevent races between the tracepoint being enabled and the check being seen.

The advantage of using the trace_<tracepoint>_enabled() is that it uses the static_key of the tracepoint to allow the if statement to be implemented with jump labels and avoid conditional branches.

> **Note**
>
> The convenience macro TRACE_EVENT provides an alternative way to define tracepoints. Check http://lwn.net/Articles/379903, http://lwn.net/Articles/381064 and http://lwn.net/Articles/383362 for a series of articles with more details.

If you require calling a tracepoint from a header file, it is not recommended to call one directly or to use the trace_<tracepoint>_enabled() function call, as tracepoints in header files can have side effects if a header is included from a file that has CREATE_TRACE_POINTS set, as well as the trace_<tracepoint>() is not that small of an inline and can bloat the kernel if used by other inlined functions. Instead, include tracepoint-defs.h and use tracepoint_enabled().

In a C file:

```
void do_trace_foo_bar_wrapper(args)
{
        trace_foo_bar(args);
}
```

In the header file:

```
DECLARE_TRACEPOINT(foo_bar);

static inline void some_inline_function()
{
        [..]
        if (tracepoint_enabled(foo_bar))
                do_trace_foo_bar_wrapper(args);
        [..]
}
```