LINUX TRACE TOOLS

Understanding the deep roots of new-age kernel instrumentation

Submitted by
Tarun Sharma
Sharath Koday

Under the guidance of
Dr Yann Hang Lee
Index

I. Abstract
II. Introduction
III. Architecture
IV. Tracepoints
V. Analysis
VI. Challenges
VII. Conclusion
VIII. References
1. Abstract
This project report portrays the idea of kernel level tracing from the perspective of a next generation tracing tool, LTTng that succeeds the previous generation tool, LTT. The main topics discussed here includes an overview of the tool, followed by a comprehensive explanation outlining the importance of each module and how they interoperate in providing strategic instrumentation to record kernel level information. Further this report describes the practical study conducted to understand the feasibility of this tool in providing real time data such as events generated by a PS2 mouse. The usefulness of the command line interface in controlling the tracepoints is presented here which is extended to the various types of events that can be logged together to give the overall picture of the process. The importance of interpreting the trace data is corroborated by an elaborate insight into the available tools that are constantly evolving-LTTng's trace analyzer and the Trace Monitoring Facility embedded inside Eclipse IDE that offers refreshing graphical analysis of the trace information through the combination of text and user interface.

2. Introduction
As Linux kernel undergoes constant modification catering requests of demanding programmers and end-users, the need for performance measurement tool is quite evident. The need for tracing has evolved too with the increase in use of more complex computer systems. An important aspect of tracing nowadays is to capture critical events in real time and high performance computing applications. Hence one of the goals of an essential tracing tool must be to incur as less overhead as possible without compromising on the amount of trace data generated. Trace tools are not new in the Linux arena, with the first trace tool LTT developed in the early 90's. Since then the features and tracing capabilities have evolved with the growing kernel.

Linux Trace Tools “new generation” (LTTng) with new added features such as User-space Read Copy Update, cache impact minimization seems as the most promising tool so far. It comes integrated with Kprobes which extends its capabilities allowing dynamic instrumentation. LTTng focuses on a capturing the system behavior enabling the end-users to zero-in on the root cause of performance lag or latency which could be due to several factors such as I/O subsystem response, scheduling and memory management techniques. This is achieved by the feature of kernel-wide instrumentation and its lock-less concurrency management algorithms. In a constant effort to keep the LTTng module as independent as possible, low system overhead is achieved even with intense event capture.

Although printf and printk are widely used for simple instrumentation, being blocking instructions, they do not offer the needed performance for capturing interrupts. LTTng, on the other hand not just provides the facility to capture Non mask-able interrupts but also achieves by logging it with high precision time stamps, thus making it suitable for real-time applications. Here, the significance of tracing must be understood well when compared to traditional logging of data. While logging provides list of all the events that have occurred, a tracing tool such as LTTng records events that occur much more frequently such as function calls, system calls, context switching, IRQ interrupt handling.

LTTng stores the trace data in the Common Trace Format represented in compressed binary format which has been designed specifically to suit tracing tool interoperability of the embedded, high-performance and Linux kernel communities due to the increased tracing speed and compactness. The storage format allows discarding the oldest information while the buffers keep updating over time.

3. Architecture
LTTng Architecture and Important Characteristics
The LTTng low level architecture consists of three core areas namely: the command line interface which defines and initiates probes that activate the tracepoints, the consumer daemon lttd which keeps waiting for the trace data to write into the local disk and the lttcore that is responsible for controlling the kernel tracing.

The core consists of several facilities that perform tasks ranging from detection for cycle counter overflows to generating events to describe the state of the kernel during start time. [2]

Before the start of the tracing activity, a session is created through the command line interface (CLI). Lt For this purpose, lttnng-tools provides a session daemon that performs the task of a session-registry.
The trace data is stored in circular “Flight-recorder” memory buffers known as RelayFS which is one of the distinct features added in the 2.0 version. The buffers contain sub-buffer that log trace data pertaining to each channel created.

Lttnd is a consumer daemon that polls these buffers and retrieves the data using an ioctl to the RelayFS and writes it to the local disk. Currently, the lttnd companion process also make use of network communication, like secured shell transfer to communicate the data to the user space by establishing remote connection between the host and the target.

**Interpreting Kernel Trace using Trace Monitoring Facility**

Trace-monitoring facility also called as “LTTng plug-in” is an Eclipse plug-in currently in development to view LTTng kernel and userspace. The framework provides a number of graphical, statistical and text based views for traces. Furthermore TMF has the ability to show concurrently the events of several traces that were recorded simultaneously.

The plug-in provides the user various conveniences to add the traces to a library. Although the traces can viewed by manually adding the Trace files in Common Trace Format directly to the project, the control of the traces can be performed through an ssh connection to the host side. Although there exists various other trace viewers such as babeltrace and LTTV, Eclipse is widely used due to the following advantages.
Eclipse plug-in supports large amount of traces. A recent upgrade has updated the buffer size to 15GB.

TMF supports translating multiple time-ordered traces.

The accuracy of LTTng can be appreciated by the high resolution trace that can be obtained (in the order of nanoseconds).

Efficient searching, filtering and bookmarking of events is possible which are found to be deficient in other trace tools.

**Instrumentation infrastructure**

Lttng offers broad range of flexibility to add tracepoints to kernel and userspace. Static tracepoints are probes that are inserted into the kernel code manually and can be activated during run-time to record information about program execution. System administrators who are well aware of the kernel subsystem benefit largely from such facility. Lttng makes use of an improvised version of kernel markers, known as Tracepoints. When the program execution encounters an enabled tracepoint, the handler associated with the tracepoints is invoked. Upon completion of the handler, the execution resumes. Another static instrumentation mechanism is the function tracer (FTrace) that keeps a check on the entry and exit of functions while bearing no over-head to the overall system. Static tracepoints are preferred when it comes to real time application debugging because of minimal overhead. The following snippet will summarize the above idea. A static tracepoint module snippet is provided in the appendix section.

```c
if (tracepoint_1_activated)
    (*tracepoint_1_probe)(arg1, arg2);
```

The main drawback of static instrumentation is the inability to add probes at run-time. When the tracepoints are inserted, they remain in the code even when deactivated. To overcome the code footprint dynamic instrumentation was introduced with the support of IBM developed Kprobes. It provides the ability of adding probes by means of a interactive command line interface. Once added and registered, each probe will be associated with a pre-handler, which gets called before the probed instruction and a post-handler which gets invoked after. Although Kprobes are efficient in terms of the size of the system dump it provides, it incurs a lot of overhead since it uses breakpoints and exception handlers.

**4. Tracepoint**

Note: These tracepoints are defined in the kernel headers but to use them, one needs to insert probes into the source code. These are not already inserted into the source code because it slows down the compilation process and therefore is added during the kernel debugging process.

A new trace-point is defined by using `TRACE_EVENT()`. The macro is defined as:

```c
TRACE_EVENT(name, proto, args, struct, assign, print)
```

where,

- name- name of the new trace-point
- prototype- callback prototype (there can be more than one callback attached to a tracepoint)
- args- arguments to be passed on to the tracepoint prototype.
- Struct (optional)- to store the data by the tracer to be passed onto the tracepoint
- assign- used to assign data to the structure (usually by 'memcpy')
- print- to print the structure in ASCII format.

Example: The usage is demonstrated using the example of the tracepoint `sched_switch` event in the kernel header.
and the source code.

4.1 The header files

The format for the TRACE_EVENT() macro in the header file must be the same as defined in the beginning of this section.

TRACE_EVENT(sched_switch,  
TP_PROto(struct task_struct *prev,  
        struct task_struct *next),  
TP_ARGS(prev, next),  
TP_STRUCT__entry(  
    __array( char, prev_comm, TASK_COMM_LEN )  
    __field( pid_t, prev_pid )  
    __field( int, prev_prio )  
    __field( long, prev_state )  
    __array( char, next_comm, TASK_COMM_LEN )  
    __field( pid_t, next_pid )  
    __field( int, next_prio )  
),

TP_fast_assign(  
    memcpy(__entry->next_comm, next->comm, TASK_COMM_LEN);  
    __entry->prev_pid = prev->pid;  
    __entry->prev_prio = prev->prio;  
    __entry->prev_state = __trace_sched_switch_state(prev);  
    memcpy(__entry->prev_comm, prev->comm, TASK_COMM_LEN);  
    __entry->next_pid = next->pid;  
    __entry->next_prio = next->prio;
),

TP_printk("prev_comm=%s prev_pid=%d prev_prio=%d prev_state=%s%s ==> next_comm=%s next_pid=%d next_prio=
%d",
    __entry->prev_comm, __entry->prev_pid, __entry->prev_prio,
    __entry->prev_state & (TASK_STATE_MAX-1) ?
        __print_flags(__entry->prev_state & (TASK_STATE_MAX-1), ",
        [1, "S"], [2, "D"], [4, "T"], [8, "t"],
        [16, "Z"], [32, "X"], [64, "x"],
        [128, "W"]) : "R",
    __entry->prev_state & TASK_STATE_MAX ? ":+" : "",
    __entry->next_comm, __entry->next_pid, __entry->next_prio)
);

source: /usr/src/linux-3.6.8/include/trace/events/sched.h

The top of the header file having the TRACE_EVENT() contains:

#undef TRACE_SYSTEM
#define TRACE_SYSTEM sched  
#if !defined(_TRACE_SCHED_H) || defined(TRACE_HEADER_MULTI_READ)  
#define _TRACE_SCHED_H

Some key observations:

① The headers for the sched_switch event contains header like "sched" and _TRACE_SCHED_H but for the other events these parameters will change accordingly.

② To include this file more than once, the header TRACE_HEADER_MULTI_READ is used otherwise the processing of the TRACE_EVENT() macro gives errors.

③ The TRACE_SYSTEM is also a very important parameter and is to be kept outside of the #if guard.

④ The keyword after #define TRACE_SYSTEM (sched here) defines the groups the TRACE_EVENT() into it. And this would be where the events will be kept inside the debugfs tracing/events directory. Due to this
grouping events as whole in chunks of groups can be enabled or disabled.

Now the only thing remaining is to include headers that are required for the TRACE_EVENT() macro and the header <linux/tracepoint.h> in the header file. After this, define the tracepoints using the macro TRACE_EVENT() with comments added on top of it describing the tracepoint. And one more thing, the bottom of the kernel file include/trace/events/sched.h should include trace/define_trace.h outside the #endif guard:

```c
#include <trace/define_trace.h>
```

### 4.2 Using the tracepoint

Prior to using the tracepoints, some guidelines are:

- One (and only one) C file must define CREATE_TRACE_POINTS in the header. The reason for including just once is explained below.
- The file using the tracepoint should include the trace header, make sure that CREATE_TRACE_POINTS is defined.

The above two things will cause the necessary functions to be created for the tracing event.

```c
#define CREATE_TRACE_POINTS
#include <trace/events/sched.h>
```

The reason why CREATE_TRACE_POINTS is included just once is because otherwise if included in more than one file, it causes linker errors. An example from the kernel, file kernel/fork.c contains only the header file:

```c
#include <trace/events/sched.h>
```

Last but not the least, the tracepoint is inserted into the code the same way it was defined, as explained earlier in the TRACE_EVENT() macro:

```c
static inline void
context_switch(struct rq *rq, struct task_struct *prev,
               struct task_struct *next)
{
    struct mm_struct *mm, *oldmm;
    prepare_task_switch(rq, prev, next);
    trace_sched_switch(rq, prev, next); //The kernel source code is free of such entries because it slows down the
                                          //compilation process, just used for kernel debugging purpose
    mm = next->mm;
    oldmm = prev->active_mm;
    ...
}
```

Source: /usr/src/linux-3.6.8/kernel/sched/core.c

### 4.3 Format file

The format file of the sched_switch has the following format:

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>sched_switch</td>
<td>ID: 33</td>
</tr>
<tr>
<td>format</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
type: unsigned short common_type; offset: 0; size: 2;
type: unsigned char common_flags; offset: 2; size: 1;
type: unsigned char common_preempt_count; offset: 3; size: 1;
type: int common_pid; offset: 4; size: 4;
```
4.4 Log

The output in the log whenever a sched_switch event occurs is of the form (taken from the lttng view command):

```
[04:37:30.629091416] (+0.000050732) sched_switch: { cpu_id = 2 }, {
  prev_comm = "swapper/2",
  prev_tid = 0,
  prev_prio = 20,
  prev_state = 0,
  next_comm = "lttng",
  next_tid = 8347,
  next_prio = 20 }
```

5. Analysis

The aim of our analysis is to read the data that the LTTng collects during the event of a mouse click and then to interpret it to see the flow of events. The interrupt number that is assigned to a PS/2 mouse is 12. It is fixed in the system. To confirm this, in a terminal type the following command:

```
$ cat /proc/interrupts
```

this will give a detailed list of devices connected to the system. The detail includes the interrupt number as well as shown below:

```
i8042 is the interrupts controller for the PS/2 mouse. This confirms the interrupts number for our device.
```

Now, we will run the LTTng tool by issuing the following commands in a terminal:

```
$t@t-Studio-1458:~$ lttng create mySession
Session mySession created.
Traces will be written in /home/t/lttng-traces/mySession-20121210-174715
$t@t-Studio-1458:~$ lttng enable-event -a -k
All kernel events are enabled in channel channel0
$t@t-Studio-1458:~$ lttng start
Tracing started for session mySession
$t@t-Studio-1458:~$ lttng stop
Tracing stopped for session mySession
$t@t-Studio-1458:~$
```

After executing the above commands, LTTng will generate log in the binary format at the mentioned location.
There are three ways to view this log:
1. Using babeltrace
2. Using LTTviewer (lttv)
3. By importing the log into the eclipse having the TMF extension added

For this study, we have used the third option, i.e., we are viewing the data in eclipse (with TMF extension). The following screenshot (Image-1) shows the point in the log starting from the point the IRQ12 is received and follows from there.

The log has a lot of log points, so to narrow down to the point of the purpose, we follow the IRQ12 line in the viewer (Image-1) and read the sequence of events from there on (Image2):

1. **irq_handler_entry**: (name=i8042, IRQ=12)
   - This is the PS/2 mouse i8042_interrupt() function call inside the kernel received by the i8042 interrupt controller.
2. **sys_unknown**: (id = 18)
   - This is a sys_stat system call which reads the file information
3. **sys_writev (fd=4)**
   - Writes the buffer data into the file described by the fd
4. **sys_read**
   - Reads the data into the buffer from the file with fd=4

![Image-1](image-url)
6. Challenges

There are a few things one should be wary about the LTTng tool. This tool is still in development stage, so all the features required for the analysis are not there, but one has a work around for that. Like in our study, we are analysing the logs that has interrupts, system calls and many other events. In the current version of the tool, not all the system calls for the x86 architecture are defined, but since it presents the id of the system call, one can somehow interpret the those “unkown” system calls.

Similarly, there are other system calls for which interpreting the data is a very difficult task. So, to make sense out of the data that one gets from the LTTng viewer, we need another helping tool like ftrace or system tap which are much more matured and evolved tools. The PS/2 mouse log using systemTap gives:

- `0xc047fbb0 : evdev_event+0x0/0x90 [kernel]`
- `0xc047f9d2 : input_pass_event+0xb2/0xc0 [kernel]`
- `0xc047c3e2 : input_handle_event+0x82/0x440 [kernel]`
- `0xc047878 : input_event+0x68/0x80 [kernel]`
- `0xc0481d35 : atkbd_interrupt+0x645/0x6a0 [kernel]`
- `0xc047f7d : serio_interrupt+0x3d/0x90 [kernel]`
- `0xc04782a1 : i8042_interrupt+0x1d1/0x330 [kernel]`
- `0xc01ad970 : handle_IRQ_event+0x60/0x140 [kernel]`
- `0xc01af9a1 : handle_edge_irq+0xb2/0xc0 [kernel]`
- `0xc0104bdf : handle_irq+0x1f/0x30 [kernel]`
- `0xc05ceedb : do_IRQ+0x48/0x1c0 [kernel]`
- `0xc017007b : alloc_pid+0x19b/0x360 [kernel] (inexact)`
- `0xc03a2cc4 : acpi_idle_enter_bm+0x262/0x291 [kernel] (inexact)`
- `0xc04b717d : cpuidle_idle_call+0x8d/0x110 [kernel] (inexact)`
- `0xc0101e9a : cpu_idle+0x5a/0x90 [kernel] (inexact)`
- `0xc05c24ba : start_secondary+0x17f/0x1ec [kernel] (inexact)`


This above log precisely shows what is happening inside the kernel when a PS/2 mouse interrupt is received.
7. Conclusion:

LTTng is a pervasive tool required to analyze the core functions of the linux kernel. Inspite of being in the development stages, this tool provides comprehensive information about the processes, functions and system tasks. In our study, we have experimented with the integration of LTTng along with SystemTap and Eclipse LTTng viewer to get a sensible trace of how the PS2 mouse communicates with the linux kernel. Given an opportunity the study would extend to remove the dependencies of LTTng with other tracing packages such as SystemTap and FTrace while at the same time hope for a much better trace interpreter that would provide information beyond high level events.
8. References:
   1. http://lwn.net/Articles/379903/
   2. Mathieu Desnoyers Michel R. Dagenais. The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux
   4. Linux kernel source code