Abstract

Debian GNU/Linux with HP Telco Extensions (Debian TE) includes a kernel tracing facility called the Linux Trace Toolkit (LTT) which allows one to record an event trace from the kernel and analyze that trace with both a graphical viewing tool and traditional Linux text tools. This paper presents introductory information about this facility and provides a tutorial introduction to analyzing data so gathered.

Introduction

Debian GNU/Linux with HP Telco Extensions (Debian TE) comes complete with the 0.9.5a version of the Linux Trace Toolkit (LTT) which consists of a kernel patch, a kernel module, and both command line and graphical user space analysis tools.

LTT provides a method and tools for acquisition of system event data and analysis thereof. There are a number of default system events defined and placed into the kernel by the LTT kernel patch. Custom events are possible too, but implementation is beyond this application note. When the kernel encounters one of these defined event locations, the event data is sent through a kernel module to a userspace daemon that in turn records it in a file on the filesystem. The database of events so recorded can then be analyzed either with the supplied LTT graphical application or with traditional text processing tools.
This application note covers basic data acquisition with LTT, conversion of this data into readable format, and analysis of this data by both traditional text processing tools as well as the supplied LTT graphical user interface application.

Packages and Installation

The kernel patch and module are, of course, integrated into the kernel that you are running, but making use of LTT will require that you have the appropriate LTT userspace package installed. The current package for LTT in Debian TE is called ltt-0.9.5+zcgl and is located in the same place from where you installed Debian TE. Installation of this package can be done with a simple `apt-get install ltt-0.9.5+zcgl` command.

This package installs the following 6 components on your system:

- **tracedaemon**: the userspace LTT daemon that communicates with the kernel patch and module
- **tracevisualizer**: the graphical trace file viewer and binary trace translator
- documentation in html format and a general man page
- helper scripts for activating LTT, generating data, viewing data graphically and translating it to ASCII (all of these begin with the `trace` prefix)
- header files and sample C code for creating custom a daemon and trace processing software
- a helper script called `tracecreatedev.sh` which will (re)create the `/dev/tracer` and `/dev/tracerU` device files for you should you need to

As part of installation, the package also creates the `/dev/tracer` and `/dev/tracerU` device files for communication with the LTT kernel module. LTT creates the device major numbers dynamically: You can find out what those are by issuing the `cat /proc/devices | grep tracer` command. Usually the tracer device files created at package installation time will work for you without modification; however, if you make use of other drivers on your system that also dynamically allocates major numbers, then the tracer device files can become stale. Should this happen (i.e., you cannot communicate with the LTT module), you can issue the `tracecreatedev.sh` command (as root of course) to re-create those files and point them to the proper major numbers.

After installing this package, you are then ready to activate LTT, gather trace data and analyze it. Note that if the tracedaemon is not running, no data is being collected and there is virtually no detectable overhead from the LTT kernel patch.

Generating Data

Data collection begins and is transferred to userspace when the LTT kernel module tracer gets loaded and the LTT daemon tracedaemon connects to it via the device files.

To help you with activating and stopping this process, LTT comes with the following helper scripts (You can, of course, use the tracedaemon command directly, but for the purposes of this application note, we will use the helper scripts.):

```
trace <seconds> <filename>
```

Activate the tracing daemon for the specified number of seconds and save the results into two files: `filename.trace` which contains the trace data, and `filename.proc` which contains the `/proc`
information at the start of the run. Both of these files are used by the tracevisualizer software to display the trace with the GUI viewer and translate it to ASCII for further manual processing.

**tracecore <seconds> <filename>**
Activate the tracing daemon for the specified number of seconds with the same output files as the `trace` command, but only trace a subset of events that are considered "core" events. These are: START (trace start); SYS_ENTRY (system call entry); SYS_EXIT (system call exit); TRAP_ENTRY, TRAP_EXIT, IRQ_ENTRY (interrupt entry); IRQ_EXIT (interrupt exit); SCHED (schedule change); KTIMER (kernel timer); SIRQ (soft IRQ management).

**tracecpuid <seconds> <filename>**
Activate the tracing daemon for the specified number of seconds with the same output files as the `trace` command, and also include CPU identification with each trace. By using this start script, each trace will be identified as belonging to the CPU on which it was executed.

**traceu <filename>**
Activate the tracing daemon for an indefinite time with output and save the results into two files: `filename.trace` which contains the trace data, and `filename.proc` which contains the `/proc` information from the start of the run. You will have to manually kill the daemon when you want data collection to stop. Use the `pkill tracedaemon` command to kill the `tracedaemon`.

### Starting Data Generation

In order to start generating some data from our system, we first have to make sure that the LTT kernel module is loaded. You do this with the following command:

```bash
# modprobe tracer
```

This loads the LTT kernel module but does not start tracing. Tracing only happens when the `tracedaemon` is activated. You do this next with the following command, which will trace all system events for 3 seconds:

```bash
$ trace 3 output1
```

This `trace` command will output the trace data into two files: `output1.trace` and `output1.proc` for the 3-second run. Both of these files will be necessary to interpret the data both graphically and textually as described in the section called “Using the GUI for Data Interpretation” and the section called “Data Interpretation with ASCII Tools”.

**Note**

Ordinarily, this is enough to start acquiring data. However, if you load drivers other than the LTT driver (which allocate their device numbers dynamically) first, then the device files that connect to the LTT driver may become stale. The symptom of this happening is that the above command will fail. Should this happen, execute the `tracecreatedev.sh` command (with no arguments) to recreate the LTT device files in the `/dev` directory. Afterwards, restart data acquisition with the above `trace` command.

If you are interested in determining from which CPU each trace is generated, then this method of starting LTT will
not produce the expected results. Instead use this script to start data acquisition:

```
$ tracecpuid 3 output1
```

This `tracecpuid` command will preserve the CPU identification for each trace event in the resultant trace files which will be visible when the trace file is converted to ASCII. This is generally a good idea, since if CPU ID information is not present in the trace, then it can be difficult to understand on SMP systems.

**Stopping Data Generation**

Ordinarily when you use the scripts described above, the data acquisition stops automatically after the specified amount of time. If you use the `traceu` script described above, however, tracing will continue indefinitely. In this case, you must manually kill the LTT daemon when your experiment is over and you want to analyze the data. Do that with the following command:

```
$ pkill tracedaemon
```

The `pkill` command does a regular expression search for a process and sends a SIGTERM signal to the resultant PID. In this case, the regular expression specifies `tracedaemon` explicitly. The `pkill` command can be found in the `procps` package.

If you use the `tracedaemon` command explicitly, then it is the `-t` option that controls the timed data acquisition. Thus, if you omit the `-t` parameter on your invocation of `tracedaemon`, the system will trace until you kill the `tracedaemon` command manually as described above.

You may want to investigate using the `tracedaemon` command directly. There are a number of options that control data acquisition. You can limit to PID or group PID and then limit which events to trace (to minimize the trace file, for example). Please see the online LTT HTML help beginning with the `/usr/share/doc/ltt/Help/index.html` file.

**Analyzing Processes**

In the section called “Generating Data” we saw how to start and stop data acquisition by using the supplied helper scripts, with the result being either a timed acquisition for the specified number of seconds or an unbounded acquisition until the daemon is killed by hand.

Sometimes it is useful to acquire a trace while a particular process is running. Unfortunately, the LTT architecture does not provide for bounding acquisition by process creation and destruction. Some minor manipulations must be used to get this effect.

There are three basic methods for doing this:

- manual control of tracing by issuing an unbounded trace command in a separate shell on the system before the process of interest is started; then, upon process termination, killing the LTT daemon. Note that this method is fine for a one-time measurement or similar situation, but it does not lend itself very well to automation.
Application Note: Using LTT for System Analysis on Debian TE

- issuing a timed trace with either the -t to the *tracedaemon* command or by using one of the time bounded helper scripts mentioned in the section called “Generating Data”. This approach is rather fuzzy around the boundary conditions but does allow for automating acquisition.

- issuing an unbounded trace command, executing the process of interest, and then using the *pgrep* (or equivalent) command to kill the *tracedaemon* once the process has exited. This is mostly useful for automation purposes.

There are several things to keep in mind when using LTT to analyze processes.

First of all, there is a certain startup time associated with the *tracedaemon*. This accounts for the tracing module to start the trace and start saving trace data to make it available for the *tracedaemon*. For automation purposes, a wait (sleep) of about 0.2 seconds is sufficient to pass this startup time; otherwise, events from the startup of the process that interests you may not be recorded.

Secondly, and this is more important for automation purposes, the *tracedaemon* “daemonizes” itself when run (the traditional double *fork* and *exec*). Thus, you cannot do a *wait* for it, nor can you check $? for its exit status in a script.

### A General Purpose Automation Script

Taking all of this into account, perhaps the best way to code a general purpose automation script for tracing while a process of interest is running is shown in the following Example 1, “A General Purpose Tracing Script”.

**Example 1. A General Purpose Tracing Script**

```bash
#!/bin/bash
# Activate tracedaemon in unbounded mode and logging cpu id's
t tracedaemon -c /dev/tracer output-$$.trace output-$$.proc
# Wait for start up time of daemon
sleep 0.2
# Run this shell's commandline arguments
$@
# Wait at end for good measure
sleep 0.1
# Stop tracing and thus generate output
pkill tracedaemon
# Output some info
echo "Output written to: output-$$.trace and output-$$.proc"
echo "Process of interest is most probably: $(( $$+3 ))"
```

The Example 1, “A General Purpose Tracing Script” script does the following 6 things:

- The *tracedaemon* command is set up for unbounded operation and CPU ID tracking.
- The script then sleeps for 0.2 seconds to pass the start up time for the *tracedaemon*. 
• The script then executes all of the its commandline parameters, thus you can use the script in the same fashion as the /bin/time command, for example.

• After the executed parameters have finished, the script waits for 0.1 seconds to possibly catch any final interactions.

• Next, the script kills the running tracedaemon with a pkill command. This will generate the output data files.

• And finally, the script outputs the name of the generated data files and a guess as to the PID of the executed command.

These resulting trace files can now be used to view and analyze the trace experiment as described in the section called “Using the GUI for Data Interpretation” and the section called “Data Interpretation with ASCII Tools”. Note that these trace files are transportable and viewable on machines other than where they were generated. Simply copy the files over; however, make sure to stick with the same architecture for the different machines (i.e., you cannot generate a trace on the Itanium and then view it on an i386 machine).

Using the GUI for Data Interpretation

The data generated from LTT tracing comes in tuples of files called filename.trace and filename.proc for every experiment. These files contain the binary trace and contents of /proc at the time of the experiment, respectively. Actually, the files do not have to be called by those names, but that is how the helper scripts create and use them, so we will stick with that convention.

The second half of the LTT tools allows viewing and analysis of this data. There are two methods available: traditional textual methods, which are covered in the section called “Data Interpretation with ASCII Tools”; and visualization through a graphical LTT application, which is covered in this section.

The command for activating all viewing options is called tracevisualizer. There is a helper script included called traceview which takes a filename with no extension as a parameter (and expects to find two files named filename.trace and filename.proc) and executes the tracevisualizer command on those files. So, for example, if we were to create a simple trace file for 5 seconds and then view it with this tool, the following commands could be used:

```
$ trace 5 myoutput
$ traceview myoutput
```

Basic Functionality of the LTT GUI

The graphical trace viewing utility (tracevisualizer) has some basic functionality for viewing the trace, listing event summaries by process and listing the raw trace data. It is presented with a traditional interface consisting of a menu bar, a tool button bar and a tabbed viewing area with three viewing tabs.

The components of the interface are as follows.

*The Menu Bar*

The menu bar provides for these simple functions.
**File**
The File menu provides functions for opening and closing trace files, for dumping an ASCII translation to a disk file and the Exit function.

*Note:* Dumping to an ASCII file is easier done with a commandline option to `tracevisualizer`. This is the subject of the section called “Data Interpretation with ASCII Tools”.

**Tools**
The Tools menu allows you to zoom in and out on the trace view and to go to a specific event and time frame.

**Options**
The Options menu allows you to change the colors of the display.

**Help**
The Help menu provides access to the html help installed in `/usr/share/doc/ltt/Help` and an About dialog for the GUI.

---

**The Tool Bar**
The toolbar adds some quick-access buttons for Menu items along with display modification functions. These buttons allow you to hide or show various icons and events on the display. If there are a lot of certain events, then the display can become crowded and difficult to perceive. In this case, use these buttons to clean up the display.

**The Viewing Tabs**
There are three tabs here. The first, called "Event Graph," is the main trace viewing window. The second, called "Process Analysis," shows summary event information (in textual form) of all processes that ran during the experiment. The third, called "Raw Trace," shows the trace in its raw textual form for the experiment.

The event graph shows the trace graphically, with time on the x-axis and processes on the y-axis. Every control-modifying event modifies the flow of the graph. A vertical line marks a shift of control from or to the kernel. A horizontal line marks a time lapse during which a process or the kernel was executing. Blue vertical lines are either an entry to or an exit from a system call. Gray vertical lines mark entry of exit from a trap. White vertical lines mark entry or exit by way of interrupt. Orange horizontal lines mark time spent in the kernel. Green horizontal lines mark time spent in a process. Time is marked in microseconds on the graph.

**Things to Keep in Mind when Using the GUI**

When using the GUI to view trace files, keep the following tips in mind.

*The left mouse button zooms in and the right zooms out.*
This is not a typical behavior one might expect from mouse control in such an application; however, once you get used to it, it does provide for fairly efficient trace browsing. Remember that zooming in happens around the current location of the pointer, and zooming out happens from the center of the display.

*The event graph display is not very efficient.*
For small trace files, this is not an issue; however, for large trace files (such as ones that get created when you trace for a long time), GUI response times suffer.

*Click on a process name to see intersections.*
If you click on a process name in the left pane, then a dashed line will be drawn across the graph, which makes it easier to see control intersections with that process.
Use the menu and buttons to go to a specific event or time.
   Especially for a large trace, these functions are particularly useful to find a specific event or time in the graph.

Turn off clutter if it is distracting.
   Again, on large (long) trace files, the display can turn into clutter with all the icons and whatnot all over. In that case, turn them off with the toolbar buttons for clarity.

The time index is in microseconds.
   The time values shown on the graph (and in the trace translation) are in microseconds since the epoch. The times shown are exact. Remember that this is not a time-sampled system; it is a trace, with every trace point in the kernel being recorded at the exact time that it happened.

SMP traces do not show well on the graph.
   The graph is not SMP capable. To see SMP effects, see the trace translation in the third tab (the "Raw Trace" tab). Don't forget to either supply the \texttt{-c} option to the \texttt{tracedaemon} command or use the \texttt{tracecpuid} helper script for trace acquisition. The really useful method for analyzing SMP effects is with ASCII tools as described in the section called “Data Interpretation with ASCII Tools”.

Data Interpretation with ASCII Tools

The LTT GUI is fine for viewing small trace files and great for presentations; however, as we all know, the best methods for analyzing data relies on tried and true text processing tools, whether you are an old-school user of \texttt{sed} and \texttt{awk} or part of the \texttt{perl} and \texttt{python} generation.

In order to get at the data and process it with our tools, we first need to translate the binary trace files collected with LTT into ASCII. Interestingly enough, the LTT translation utility is the same command (\texttt{tracevisualizer}) as for the LTT GUI described in the section called “Using the GUI for Data Interpretation”: The only difference is in the command line options.

This section describes the \texttt{tracevisualizer} command operation for translating trace data to ASCII and then uses a simple example to show some of the analyses possible through these means.

Translating Trace Files

The \texttt{tracevisualizer} command only enters graphical mode if the \texttt{-g} option is given; otherwise a translation of the trace is done, which is controlled by various options. Before I discuss some of these options, however, I should mention that there are some helper scripts included with the LTT package that make the use of this functionality easier.

These scripts are:

\texttt{tracedcore <filename>}
   Translate and dump only the core events from the trace file supplied as argument. (These are the same core events that are mentioned for the \texttt{tracecore} command described in the section called “Generating Data”.)

   The script expects two files to exist: \texttt{filename.trace}, the binary trace file; and \texttt{filename.proc}, the proc information. If you used a helper script to acquire the data, then these files will exist and you only need specify the basename (i.e., not the extensions). The output is written to a file called \texttt{filename.data}.

\texttt{traceanalyze <filename>}
   Perform basic analysis on the fileset specified by \texttt{filename.trace} and \texttt{filename.proc}, and output that summary into a file called \texttt{filename.data}. 

---

8
tracedump <filename>
This command translates all events in the binary trace file specified by the argument (in the same fashion as for
the other helper scripts above) and writes the ASCII output with a little summary information to a file called
filename.data.

This tracedump is the main command we will be using to translate our trace files.

The tracevisualizer Options

The tracevisualizer command has a number of options that control that which is output. As mentioned before, the
-g option is for entering GUI mode. Some of the other options have the following effects: (Please see the online
HTML documentation for more details.)

-o (omit event types)
Do not include any of the events that are given on a space-separated list following this option. The event types
can be any of the following specifiers:

- "START" - Trace start
- "SYS_ENTRY" - System call entry
- "SYS_EXIT" - System exit
- "TRAP_ENTRY" - Trap entry
- "TRAP_EXIT" - Trap exit
- "IRQ_ENTRY" - Interrupt entry
- "IRQ_EXIT" - Interrupt exit
- "SCHED" - Schedule change
- "KTIMER" - Kernel timer
- "SIRQ" - Soft IRQ management
- "PROCESS" - Process management
- "FS" - File system management
- "TIMERS" - Timer management
- "MEM" - Memory management
- "SOCKET" - Socket communications
- "IPC" - System V IPC communications
- "NET" - Network device management

-t (trace event types)
This will only include the specified events (a space-separated list of the same event type specifiers as for the –o
option above) in the output.

\texttt{-c (trace CPU ID)}

Only include events in output that occurred on the specified CPU (numeric value).

\texttt{-p (trace PID)}

Only include events that were attributable to the specified PID in output.

\texttt{-a (analyze trace)}

Analyze the trace and output the summary and translation to the output file.

There are a few other options having mainly to do with not outputting certain information. While these and the above options may be of use to limit the size of the resultant data file, in most cases this is not necessary, and it is just simpler to use the helper script \texttt{tracedump} which will provide us with the base data file to process.

### Analysis of an Example Trace

For the rest of this section, we will analyze an example program, which is simply an \texttt{ls} command issued to the system and traced with our general purpose tracing script shown in Example 1, “A General Purpose Tracing Script”.

To generate the trace files, we issue the following command (assuming, of course, that we called the script listed in Example 1, “A General Purpose Tracing Script” \texttt{runtrace}):

\begin{verbatim}
$ runtrace ls
\end{verbatim}

This generates two files, and in our case they are called \texttt{output-13629.trace} and \texttt{output-13629.proc}. The number in the filename is automatically assigned by the trace script and is the PID of the script. Our process of interest is thus 13632. (See the discussion of the script in the section called “A General Purpose Automation Script”.)

The first step in analysis is to translate this fileset into an ASCII file of the trace. We will accomplish this with the use of the \texttt{tracedump} helper script.

\begin{verbatim}
$ tracedump output-13629
\end{verbatim}

This command generates a file called \texttt{output-13629.data} which contains the ASCII translation of the trace and some summary information at the top of the file. Each trace is contained on one line; thus, the file lends itself to analysis with text processing tools.

The summary section at the top of the file reads as follows:

\begin{verbatim}
Trace start time: (1077669309, 148617)
Trace end time: (1077669309, 484552)
Trace duration: (0, 335935)
Number of occurrences of:
  Events: 5914
  Scheduling changes: 38
  Kernel timer tics: 170
\end{verbatim}
System call entries: 757
System call exits: 748
Trap entries: 494
Trap exits: 494
IRQ entries: 714
IRQ exits: 714
Bottom halves: 0
Timer expiries: 2
Page allocations: 94
Page frees: 206
Packets Out: 2
Packets In: 2

From this information we can see that the trace experiment ran for 335935 microseconds—just over 1/3 of a second. This tallies with what we expect since our tracing script sleeps for 0.3 seconds total, plus there is some time for executing the `ls` system call along with other overhead.

We also see that there were 5914 traced events, and if we do a line count on the file with this command:

```
$ wc -l output-13629.data
5939 output-13629.data
$
```

We see that this does tally with the file size. The extra lines are for the summary information at the top of the file.

A typical trace in this file looks like the following:

```
1 File system 1,077,669,309,154,477 13631 31 EXEC : sleep
```

This particular trace happens to be an `exec` filesystem trace for the sleep program. In fact, let’s see how many `exec`’s are in the file.

```
$ grep EXEC output-13629.data
1 File system 1,077,669,309,154,477 13631 31 EXEC : sleep
1 File system 1,077,669,309,365,412 13632 28 EXEC : ls
1 File system 1,077,669,309,369,318 13633 31 EXEC : sleep
0 File system 1,077,669,309,480,645 13634 31 EXEC : pgrep
$
```

You can see that there were four programs `exec`-ed during the experiment; in fact, these are what we expected
since that is what our runtrace script did. Note that the numbers after the long number are the PIDs of the processes and our process of interest is, in fact, 13632. Also note that the time between the first sleep and the next ls is:

\[
1,077,669,309,365,412 - 1,077,669,309,154,477 = 210,935 \text{ us}
\]

This tallies with our script, which slept for 0.2 seconds before executing ls. Also note that the time for the ls command is:

\[
1,077,669,309,369,318 - 1,077,669,309,365,412 = 3,906 \text{ us}
\]

That translates to 3.9 ms from exec to exec (quite fast). The time numbers, incidentally, are microseconds since the epoch. And finally, note that the first three programs were executed on processor 1, while the last was executed on processor 0.

Let’s look at SMP effects. First, how many traces were generated on each CPU? Note that the first number in each trace is the CPU ID such that the following commands yield:

```
$ grep '^1 ' output-13629.data | wc -l
2924
$ grep '^0 ' output-13629.data | wc -l
2990
```

From this we see that there were approximately the same number of trace events on both CPUs. That may be interesting but not very useful for this experiment. Let’s see if there was much CPU thrashing for ls, our process of interest. We know that it had a PID of 13632 from before and that the exec happened on CPU 1. From those facts, we can construct the following commands:

```
$ grep '13632' output-13629.data | grep '^1' | wc -l
485
$ grep '13632' output-13629.data | grep '^0' | wc -l
23
```

We see that our process 13632 spent most of its time on CPU 1 (95.5%, in fact). Let’s take a closer look at how process 13623 is created. If we edit the output-13629.data file and immediately search for 13623, we jump to the creation point and see the following traces:

```
0 Process ...412 13628 17 FORK:13632
0 Process ...412 13628 17 WAKEUP PID:13632; STATE:2
0 Syscall exit ...412 13628 8
0 Trap entry ...412 13628 24 TRAP:VHPT miss; IP:0x00000000
1 Sched change ...412 13632 28 IN:13632; OUT:0; STATE:0
```
0 Trap exit ...412 13628 8
1 Syscall entry ...412 13632 17 SYSCALL:getpid; IP:0x2000000000234042
1 Syscall exit ...412 13632 8
0 Trap entry ...412 13628 24 TRAP:VHPT miss; IP:0x00000000
1 Trap entry ...412 13632 24 TRAP:VHPT miss; IP:0x00000000
0 Trap exit ...412 13628 8

### Note

I've compressed the traces somewhat so they fit onto the page. The "...412" represents the time stamp which is 1,077,669,309,365,412 for all these traces, signifying that all of these took less than a microsecond to execute.

From this example trace, we see that process 13628 (our runtrace script) is running on processor 0 and then forks process 13632 (our ls process). Process 13632 wakes up on processor 1 shortly afterwards due to a schedule change. Our process 13623 begins life with a call to getpid (the normal routine for a fork situation). As we continue further down the trace, we see that our process executes on processor 1 while the runtrace process continues on processor 0 until it hits a wait a little later on to wait for completion of our ls process. This, then, represents a fine example of the SMP capability of the system being utilized correctly.

As a final bit of analysis on our process 13623, we can answer the question "How many system calls does an 'ls' take?" with the following command:

\[\$\;\text{grep '13632' output-13629.data} \mid \text{grep SYSCALL} \mid \text{wc -l}\]

That concludes our simple analysis of this experiment. Through the use of the LTT trace, we were able to learn quite a bit about a simple process. In fact, upon closer examination, you may notice that there really was no need to sleep before and after our ls experiment since all events ended up in the trace anyway. This is because the system was idle beyond executing our experiment. This may not be the case on heavily loaded systems. In any case, the preconditions and postconditions of our experiment are easy to filter out. We have only scratched the surface of what is possible with LTT tracing analysis, but hopefully this discussion has given you some ideas on how to analyze your own situations and perhaps answer questions that were previously difficult or even impossible to answer.

### Where to Find More Information

This short application note only scratched the surface of what is possible with LTT. More information is available on your system in the LTT documentation, which can be found starting at the `/usr/share/doc/ltt/Help/index.html` file after installing the ltt-0.9.5+zcg1 package.

Programming example code is available in the `/usr/share/doc/ltt/Examples` directory. This includes programming for custom kernel events, custom userspace events and custom trace reading software. The full source of LTT is, of course, available with the source package install.

The official LTT website is at http://www.opersys.com/LTT (which hosts official distributions, general information, user-contributed patches and mailing lists).