Tracing Time Operating System State Determination

Jean-Hugues Deschênes, Mathieu Desnoyers and Michel R. Dagenais*

Department of Computer and Software Engineering, Ecole Polytechnique, P.O. Box 6079, Station Downtown, Montreal, Quebec, Canada, H3C 3A7, Canada

Abstract: In recent years, tracing operating system behavior by recording kernel events has proven to be a particularly effective tool. However, when used to characterize the system's behavior through time, including its state, the list of state transitions that the kernel events represent is not sufficient to characterize the state for the entire data acquisition period. The initial operating system state, when tracing starts, is also required.

The challenge lies in obtaining a complete snapshot of the initial state, while minimizing the impact on the system being traced. This impact may be in terms of CPU or disk I/O consumption, instrumentation memory, or burst activity at trace start time detrimental to the real-time response. Such impact is especially disturbing on small real-time limited resources embedded systems.

In this paper, we will propose an efficient approach to extract such initial state information and discuss the software module we have developed to provide the aforementioned data to the LTTng tracing tool and its accompanying viewer, LTTV. This module not only improves LTTV's accuracy by providing the initial state of all processes in the system, but also provides an inventory of relevant kernel objects at minimal cost, without increasing noticeably the interrupt latency.

INTRODUCTION

Tracing operating system behavior through kernel events is a powerful tool for finding certain types of performance bottlenecks, or debugging complex real time interactions between various system entities, among other things. Because of the large volume of events even for short acquisition periods, a trace visualization and analysis program is a necessary part of a complete trace toolkit. This program maintains a representation of the kernel's state throughout the data acquisition period, using the events as transition points within this state.

While it is possible to rebuild part of the kernel's state at trace startup from certain types of events (for example, if we have a “syscall” event at instant *t*, then we can determine that at instant *t*-1, the system was in user mode), one cannot get a complete snapshot of the state at instant *t*=0 using only later events. Consider, for example, a process for which no event gets generated during the entire data acquisition period, as is often the case for daemons; Such a process would be missing from the state maintained by the trace analysis tool.

Other data may be voluntarily missing from kernel events, for efficiency reasons. For example, it would be unnecessarily costly to include process names in all scheduler-related events. This means that the names of the processes that were running at trace startup are not present in the trace's kernel events.

Thus, to get an accurate picture of the state at every instant (including *t*=0), it is necessary to perform an inventory of the relevant kernel objects that were present at trace startup. This needs to be done efficiently, with the smallest impact possible on the instrumented system.

Previous work

LTT

With the first releases of the Linux Trace Toolkit [1], initial operating system state was determined by navigating the */proc* directory. Relevant information was parsed by a daemon at trace startup and stored on disk. This approach has the drawback of relying on the presence of the */proc* filesystem, which may not be present in embedded systems. It also means that the daemon has to have intimate knowledge of the */proc* filesystem's structure and has to open and parse several files every time a trace is started.

Other open-source tracing systems are not designed to analyze the kernel state. The focus is either on providing fine-grained instrumentation (like KFT[2]) or on gathering statistics, for performance analysis (such as LKST's [3] lkstlogtools [4]).

Some closed source products have had to tackle this issue as well. QNX's *System Profiler* [5] uses a strategy similar to the one exposed in this paper, reporting part of the initial operating system's state through kernel events at trace startup. This tool, however, uses process and thread creation events when enumerating existing processes and threads. Thus, although it may be possible to guess that a given object was present at trace startup, one cannot have absolute certainty that it is indeed the case. Additionally, no inventory is made of other objects, such as file descriptors. Finally, we have not been able to observe any run-time events before the end of the process and thread enumeration phase, leading us to believe that the system as a whole is highly disrupted while this inventory is performed.

*Address correspondence to this author at the Department of Computer and Software Engineering, Ecole Polytechnique, P.O. Box 6079, Station Downtown, Montreal, Quebec, Canada, H3C 3A7, Canada; Tel: 1-514-340-4711; Fax: 1-514-340-5139; E-mail: michel.dagenais@polymtl.ca

1874-107X/08

Encompassing Architecture

The proposed approach, implemented in the statedump module is designed to be integrated to the LTTng [6] tracing tool. It is a modular design, bringing minor modifications to existing kernel code while adding a few modules.

From a high level point of view, as shown in Fig. (**1**), LTTng is made up of an instrumented kernel, which generates the data to be analyzed. These events are sent to the tracing module, which is responsible for trace management and data channel management. The data is then conveyed through a fast virtual filesystem to a data acquisition daemon. This daemon simply writes the data to a local filesystem (presumably residing on a disk). A control program communicates with the control module to initiate or otherwise terminate data collection.

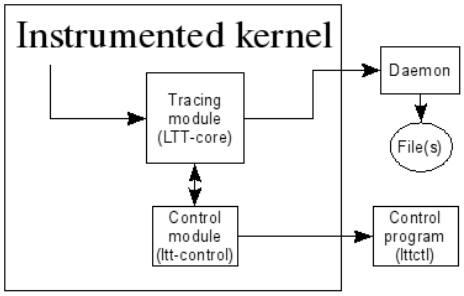

Fig. (1). LTTng architecture.

Implementation

Architecture

To adhere to LTTng's modular design philosophy and since the statedump is an optional (albeit desirable) component, it has been implemented as a kernel module. It is registered with the tracing module upon insertion and then invoked when a new trace is started.

To greatly simplify things, the statedump module navigates through relevant kernel structures and generates LTTng events, just like the kernel instrumentation. The event types are different, however, and belong to a statedump facility. They are listed in Table **1**.

Table 1. Statedump Event Types.

Event	Description
enumerate_process_state	State of all threads and tasks
enumerate_file_descriptors	Active file descriptors for all tasks
enumerate_modules	Loaded kernel modules
enumerate_vm_maps	Virtual memory areas for all tasks
enumerate_interrupts	Interrupts
statedump_end	Signals the end of state dump

The types of enumerated kernel objects were chosen for their expected relevance in understanding complex traces. Additionally, the enumerate_process_state event will assist LTTV in determining initial operating system state by providing process names and process state information. The statedump_end event marks the end of the state dumping phase and is used as a marker to indicate that beyond that point, all processes' states can be fully determined.

Special care has been taken when holding locks, the most critical of which is undoubtedly the *tasklist_lock*, since it needs to be acquired by the scheduler to complete its work. To this end, the lock is held only when moving through the task list. To prevent the *task_struct* from being freed while processes are enumerated, the structure's use count is incremented.

Processes' States

Of particular interest are the processes' state at trace startup. This includes the process type (normal or kernel thread), run status and operating mode.

The process type information can easily be inferred from the kernel's *task_struct*, looking at the memory space descriptor field, which is set to NULL in the case of kernel threads.

The run status information is obtained from *task_struct*'s state & exit_state fields, using the decision tree depicted in Fig. (**2**). Looking at this figure, one can notice that no decision is made as to which process(es) was(were) running when the inventory was performed. Was the statedump module guaranteed to run on a single processor system, it would have been trivial to identify the running process, the only possibility being the statedump module's inventory thread. On multiprocessor systems, however, it is not possible for the statedump module to know which processes, if any, are running on the processors other than the one its acquisition thread is currently running on.

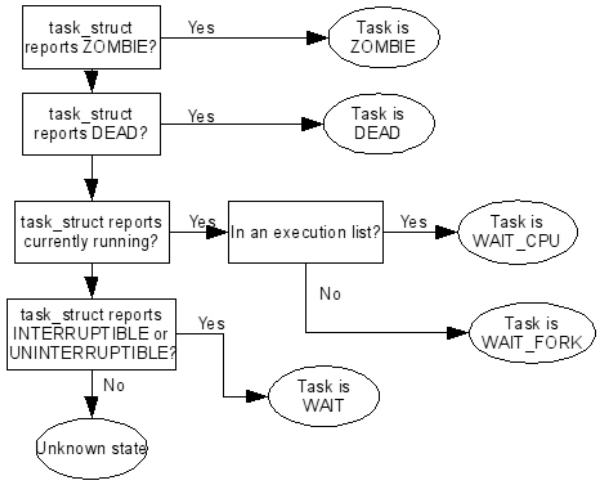

Fig. (2). Process run status decision tree.

To get around this issue, the module schedules a function in a work queue on every processor it is not running on. This function simply releases a semaphore held by the statedump thread. This way, it is guaranteed that a kernel event has been generated by every processor when each of them changes its execution flow to the aforementioned function. The visualization tool will then have all the information it needs to determine what activity was occurring on each processor when tracing began.

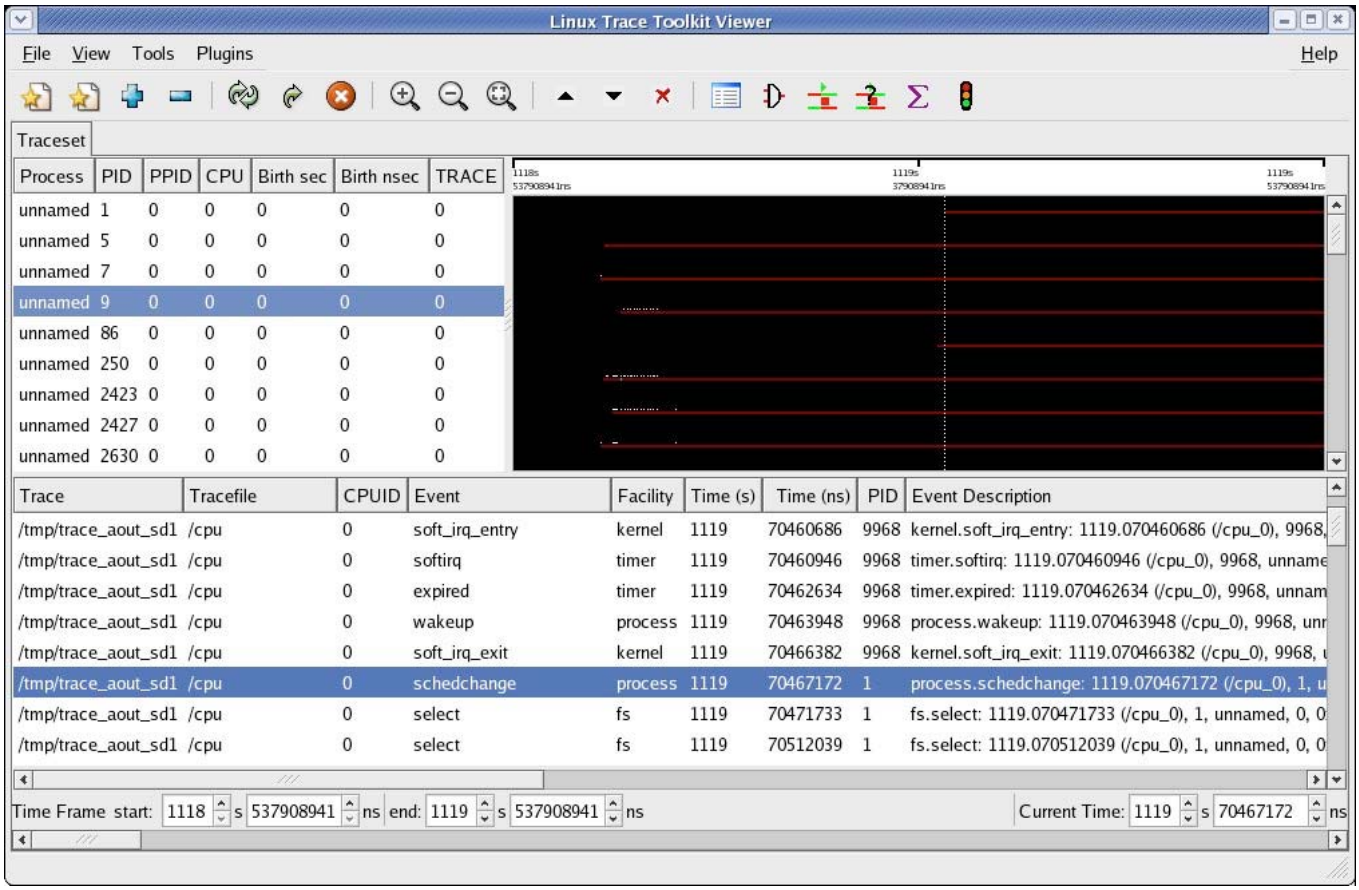

Fig. (3). LTTV without ltt-statedump events.

The per-processor scheduled function strategy will also be helpful when determining each process's operating mode, by providing the guarantee that each processor will have been at least once in a state where it was neither in a trap, nor in interrupt context. It will then be up to the visualization tool, using the various state transition kernel events, to determine what had been the processes' operating mode at trace start.

Visualization Tool Integration

To simply display the new data without using it to determine operating system state, no modification to LTTV would have been required as it is designed to use the same XML-based event descriptions as the other events. This is sufficient for *enumerate_file_descriptors*, *enumerate_ modules*, *enumerate_vm_maps* and *enumerate_interrupts* events.

For *enumerate_process_state* events, however, this is not enough since we want the information they contain to be taken into account in the kernel's state representation. Thus, in line with LTTV's design, we have added callbacks for processing these types of events when they are processed from input files. This allows to add or update processes to the maintained kernel state.

RESULTS

All results presented were obtained on a computer running Linux kernel 2.6.23.1 with a 2,8GHz Pentium4 processor.

Initial State

To assess the statedump module's benefits, we have gathered screen captures of LTTV ignoring (Fig. **3**) and taking into account (Fig. **4**) the *enumerate_process_state* events. From these, one can see how the information presented to the user is enriched with process names, parent process identifiers and the processes' creation time (considered to be the time at which the statedump module identified the process). Additionally, some processes which were simply not present in the graph (since no event associated with them were present in the trace) can now be seen.

Performance Impact

To measure the module's impact on the instrumented system, we have gathered information on global execution time increase, as well as system disturbance during data acquisition.

We have measured execution time increase for a simple reference program, which simply forks 50, 500 or 5000 times and waits for all forked processes to terminate. Each forked process performs enough iterations of a loop so that the test globally lasts approximately 10 seconds. We timed all 3 variants, with and without the statedump module, 5 times and averaged the results. To better reflect real-world conditions, a dbench session simulating 100 clients was running in the background when the results were acquired. From

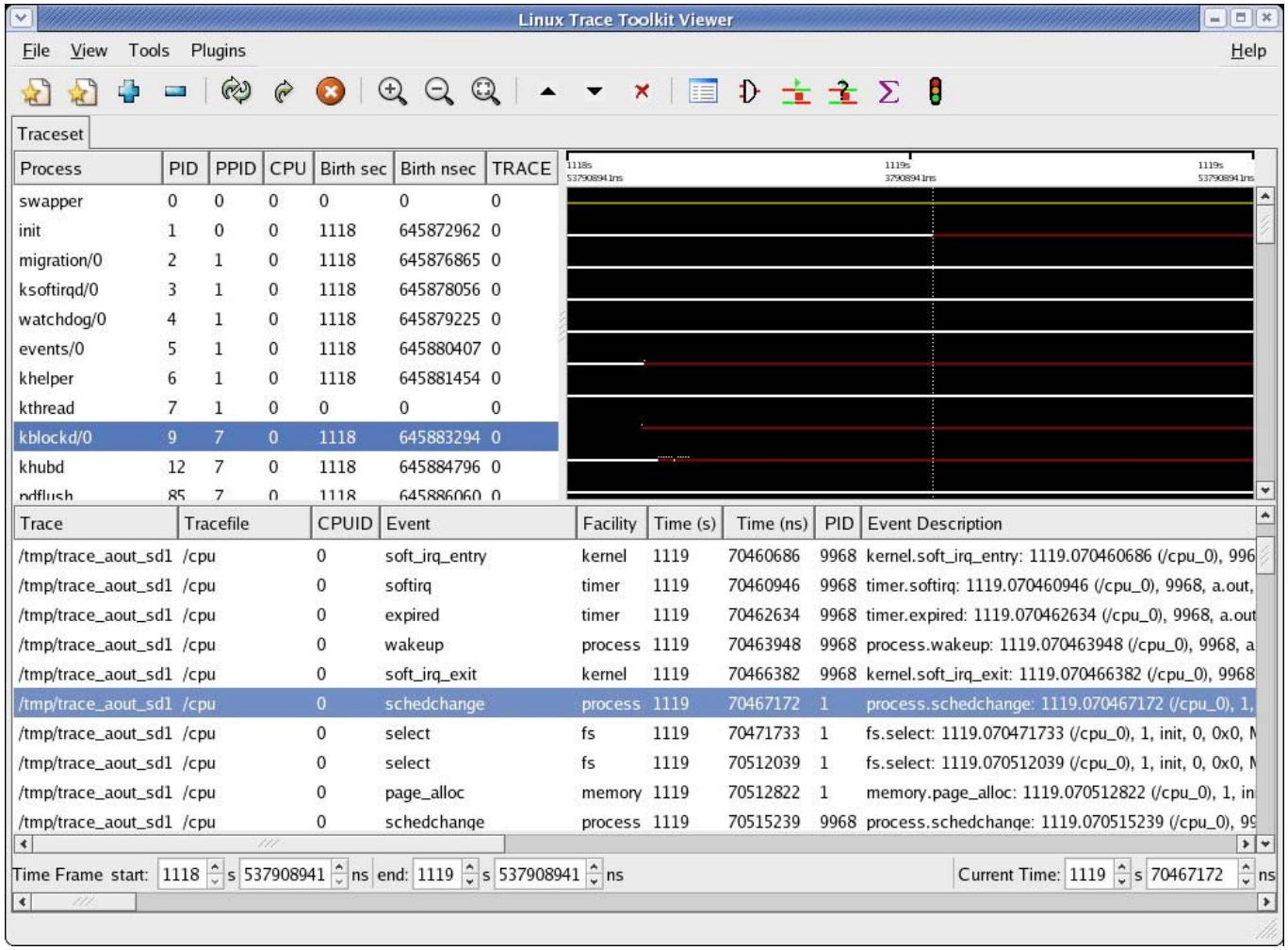

Fig. (4). LTTV with ltt-statedump events.

Table **2**, one can see that on average, the test duration is actually slightly smaller when the module is present. This means its presence in the kernel makes no statistically significant difference as far as execution time is concerned.

Table 2. Average Test Duration for a Varying Number of Forked Processes

Nb. Processes	Avg. Time with Module (Seconds)	Avg. Time Without Module (seconds)
50	11.94	12.08
500	11.81	11.85
5000	12.85	13.08

This matches our expectations. Looking at the event timestamps, the *statedump_end* event is observed on average 5.44 ms after the start of the trace, with a worst case of 8.59ms.

To measure system disturbance, we have measured the amount of time spent while holding locks. This has been done by sampling the x86 architecture's TSC register before acquiring and after releasing locks. We have instrumented the main task list lock, the task lock (in task_struct), the interrupt list lock and the tasks' file list lock. The results are shown in Table **3** for our reference program and in Table **4** for a modified version which opens 1000 files before performing the 500 calls to fork().

Table 3. Lock Hold Time for 500 Processes

Lock	Average Hold Time (Microseconds)	Maximum Hold Time (Microseconds)
Task list	0.36	18.46
Task	19.31	885.85
Interrupts	0.2	9.32
File list	14.96	246.42

These results show that the task list hold times are small and are not influenced by the amount of data acquired. This

is explained by the fact that this lock is not held while data is being acquired per se, but only when navigating the list.

Table 4. Lock Hold Time for 500 Processes, Each Having 1000 Opened Files

Lock	Average Hold Time (Microseconds)	Maximum Hold Time (Microseconds)
Task list	0.35	18.02
Task	36.94	113,804
Interrupts	0.15	11.62
File list	520.63	3,311.74

Task and interrupt hold times are also very reasonable, on average, with the task lock being held for a few tens of microseconds. This incurs a negligible impact on operating system latency.

The file list lock hold times, however, are influenced by the amount of data acquired. This was to be expected since this lock is held throughout the file inventory phase. With an average hold time of approximately 520 microseconds for processes with plenty of file descriptors, one can think that the instrumented system would be disturbed somewhat during the data acquisition phase. Taking into account that the file descriptor objects are protected by an RCU lock, we believe that the impact is far less dramatic than it would seem at first glance.

As a reference, the quantity of events generated for our test program, both without any files descriptors open and with 1000 file descriptors per process, is shown in Table **5**. Keeping in mind that *dbench* was running in the background when this data was acquired (for both cases), it is expected that similar numbers would be encountered on production systems.

CONCLUSION

Throughout this paper, we have shown how the *statedump* module integrates easily and elegantly to LTTV and LTTng to enrich the data already collected by the instrumentation. Moreover, with the additional data, relevant aspects of the system's initial state can now be fully determined.

Table 5. Total Number of Objects Enumerated for the Reference Program (with 500 Processes), with no File Descriptors and with 1000 File Descriptors Per Process

Object	No File Descriptors	1000 File Descriptors
Tasks	785	699
Vm maps	10,246	11,196
Interrupts	19	19
File descriptors	2020	112,718

All of this additional data has been acquired with a very low additional impact on the system and for a very short period of time. This way, the *statedump* module adds real benefits at minimal cost. For those rare cases where its use is not desired, LTTng and LTTV will still function properly, since their modular architecture (and the modifications we have made to them) do not depend on *statedump*.

ACKNOWLEDGMENTS

The financial support of the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

REFERENCES

[1] K. Yaghmour, and M. R. Dagenais, "Measuring and characterizing system behavior using kernel-level event logging", Proceedings of the USENIX Annual 2000 Technical Conference, San Diego, California, USA, June 2000, pp. 13-26.

[2] Tim Bird, “Learning the kernel and finding performance problems with KFI”, Proceedings of the Consumer Electronics Linux Forum International Technical Conference, Yokohama, Japan, June 2005.

[3] LKST, “Linux Kernel State Tracer”, http://lkst.sf.net/, viewed on September 18, 2008.

[4] Hitachi, “LKST Log Tools' Manual”, http://downloads.sourceforge.net/lkst/lkstlogtools-1.2.0-en.pdf, viewed on September 18, 2008, 2006.

[5] QNX, “The System Profiler User Guide”, http://www.qnx.com/developers/docs/6.3.0SP3/ide_en/user_guide/sysprof.html, viewed on September 18, 2008, 2004-2008.

[6] M. Desnoyers, and M.R. Dagenais, “Low disturbance embedded system tracing with Linux Trace Toolkit Next Generation”, Proceedings of the 2006 Consumer Electronics Linux Forum, San Jose, California, USA, April 2006.

Received: September 20, 2008 Revised: November 06, 2008 Accepted: December 05, 2008