The Frysk Execution Analysis Architecture

Andrew Cagney
Red Hat Canada Limited
cagney@redhat.com

Abstract

The goal of the Frysk project is to create an intelligent, distributed, always-on system-monitoring and debugging tool. Frysk will allow GNU/Linux developers and system administrators: to monitor running processes and threads (including creation and destruction events); to monitor the use of locking primitives; to expose deadlocks, to gather data. Users debug any given process by either choosing it from a list or by accepting Frysk's offer to open a source code or other window on a process that is in the process of crashing or that has been misbehaving in certain user-definable ways.

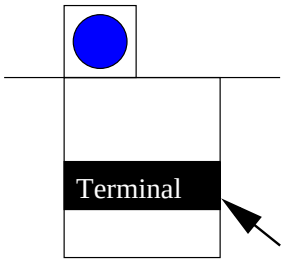
1 Introduction

This paper will first present a typical Frysk use-case. The use-case will then be used to illustrate how Frysk differs from a more traditional debugger, and how those differences benefit the user. This paper will then go on to provide a more detailed overview of Frysk's internal architecture and show how that architecture facilitates Frysk's objectives. Finally, Frysk's future development will be reviewed, drawing attention areas of the Linux Kernel that can be enhanced to better facilitate advanced debugging tools.

2 Example — K. the Compiler Engineer

K., a compiler engineer, spends a lot of time running a large, complex test-suite involving lots of processes and scripts, constantly monitoring each run for compiler crashes. When a crash occurs, K. must first attempt to reproduce the problem in isolation, then reproduce it under a command-line debugging tool, and then finally attempt to diagnose the problem.

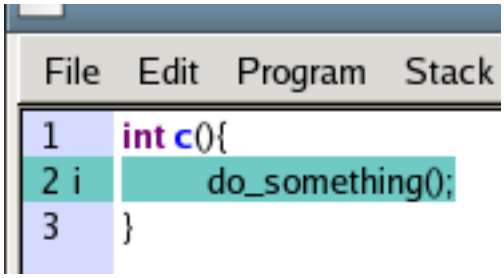
Using Frysk, K. creates a monitored terminal:



From within that terminal, K. can directly run the test framework:

```
$ ls
Makefile
$ make -j5 check
```

When a crash occurs, K. is alerted by the blinking Frysk icon in the toolbar. K. can then click on the Frysk icon and bring up the source window displaying the crashing program at the location at which the crash occurred:



3 Frysk Compared To Traditional Debugger Technology

In the 1980s, at the time when debuggers such as GDB, SDB, and DBX were first developed, UNIX application complexity was typically limited to single-threaded, monolithic applications running on a single machine and written in C. Since that period, applications have grown both in complexity and sophistication utilizing: multiple threads and multiple processes; shared libraries; shared-memory; a distributed structure, typically as a client-server architecture; and implemented using C++, Java, C#, and scripting languages.

Unfortunately, the debugger tools developed at that time have failed to keep pace of these advances. Frysk, in contrast, has the goal of supporting these features from the outset.

3.1 Frysk Supports Multiple Threads, Processes, and Hosts

Given that even a simple application, such as firefox, involves both multiple processes and threads, Frysk was designed from the outset to follow Threads, Processes, and Hosts. That way the user, such as K., is provided with a single consistent tool that monitors the entire application.

3.2 Frysk is Non-stop

Historically, since an application had only a single thread, and since any sign of a problem (e.g., a signal) was assumed to herald disaster, the emphasis on debugging tools was to stop an application at the earliest sign of trouble. With modern languages, and their managed run-times, neither of those these assumptions apply. For instance, where previously a `SIGSEGV` was indicative of a fatal crash, it is now a normal part of an application's execution being used by the system's managed run-time as part of memory management.

With Frysk, the assumption is that the user requires the debugging tool to be as unobtrusive as possible, permitting the application to run freely. Only when the user explicitly requests control over one or more threads, or when a fatal situation such as that K. encountered is detected, will Frysk halt a thread or process.

3.3 Frysk is Event Driven

Unlike simpler command-line debugging tools, which are largely restricted to exclusively monitoring just the user's input or just the running application, Frysk is event-driven and able to co-ordinate both user and application events simultaneously. When implementing a graphical interface, this becomes especially important as the user expects Frysk to always be responsive.

3.4 Frysk has Transparent Attach and Detach

With a traditional debugging tool, a debugging session for an existing process takes the form:

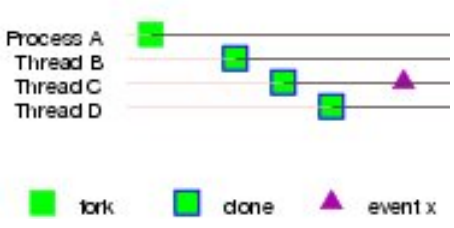
- attach to process
- examine values, continue, or stop
- detach from process

That is, the user is firstly very much aware of the state of the process (attached or detached), and secondly, is restricted to just manipulating attached processes. With Frysk, the user can initiate an operation at any time, the need to attach being handled transparently.

For instance, when a user requests a stack backtrace from a running process, Frysk automatically attaches to, and then stops, the process.

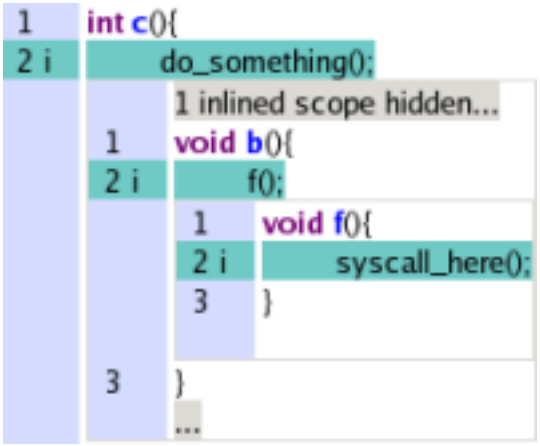
3.5 Frysk is Graphical, Visual

While a command-line based tool is useful for examining a simple single-threaded program, it is not so effective when examining an application that involves tens if not hundreds of threads. In contrast, Frysk strongly emphasizes its graphical interface providing visual mechanisms for examining an application. For instance, to examine the history of processes and events, Frysk provides an event line:



3.6 Frysk Handles Optimized and In-line Code

Rather than limiting debugging to applications that are written in C and compiled unoptimized, Frysk is focused on supporting application that have been compiled with optimized and inlined code. Frysk exploits its graphical interface by permitting the user to examine the inlined code in-place. For instance, an in-lined function `b()` with a further in-line call to `f()` can be displayed as:



3.7 Frysk Loads Debug Information On-demand

Given that a modern application often has gigabytes of debug information, the traditional approach of reading all debug information into memory is not practical. Instead Frysk, using `libelf` and `libdw`, reads the debug information on demand, and hence ensures that Frysk's size is minimized.

3.8 Frysk Itself is Multi-Threaded and Object Oriented

It is often suggested that a debugging tool is best suited at debugging itself. This view being based on the assumption that since developers spend most of their time using their own

tools for debugging their own tools, they will be strongly motivated to at least make debugging their tool easy. Consequently, a single-threaded procedural debugging tool written in C would be best suited for debugging C, while developers working on a multi-threaded, object-oriented, event-driven debugging tool are going to have a stronger motivation to make the tool work with that class of application.

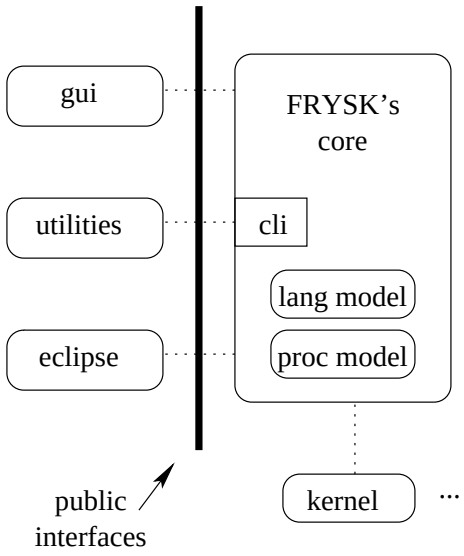
3.9 Frysk is Programmable

In addition to a graphical interface, the Frysk architecture facilitates the rapid development of useful standalone command-line utilities implemented using Frysk's core. For instance the command line utility `ftrace`, similar to `strace`, was implemented by adding a system call observer that prints call information to the threads being traced, and the program `fstack` was implemented by adding a stop observer to all threads of a process so that as each thread stopped its stack back-trace could be printed.

4 The Frysk Architecture

4.1 Overview

At a high level, Frysk's architecture can be viewed as a collection of clients that interact with Frysk's core. The core provides clients with alternate models or views of the system.



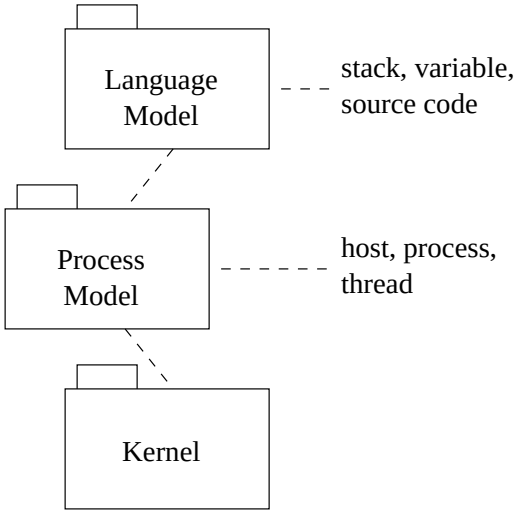
Frysk's core then uses the target system's kernel interfaces to maintain the internal models of the running system.

4.2 The Core, A Layered Architecture

Aspects of a Linux system can be viewed, or modeled, at different levels of abstraction. For instance:

- a process model: as a set of processes, each containing threads and each thread having registers and memory
- a language model: a process executing a high-level program, written in C++, having a stack, variables, and code

Conceptually, the models form sequence of layers, and each layer is implemented using the one below:

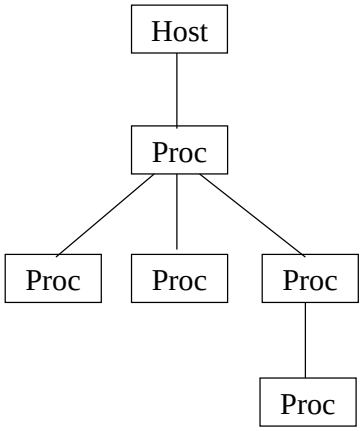


For instance, the language model, which abstracts a stack, would construct that stack's frames using register information obtained from the process model.

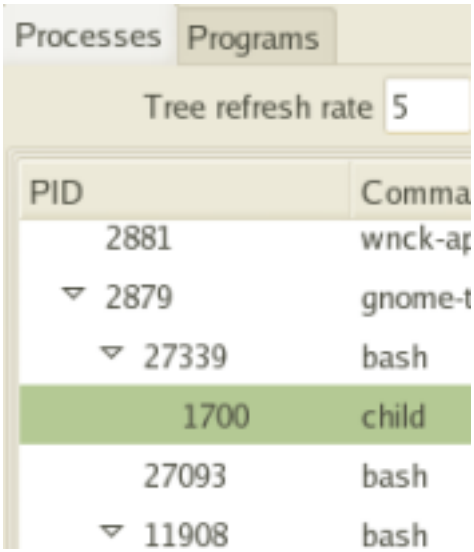
The core then makes each of those models available to client applications.

4.2.1 Frysk's Process Model

Frysk's process model implements a process-level view of the Linux system. The model consists of host, process, and task (or thread) objects corresponding to the Linux system equivalents:



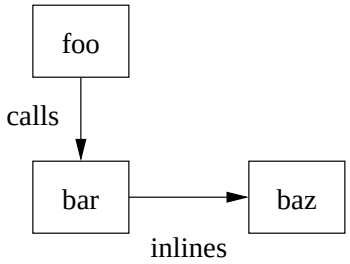
Frysk then makes this model available to the user as part of the process window:



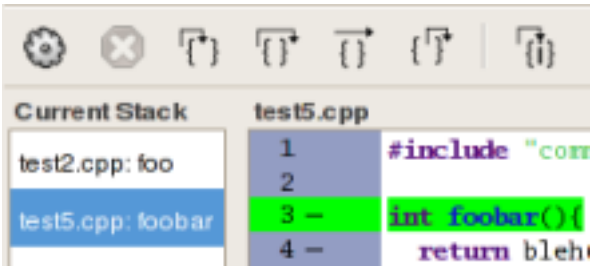
When a user requests that Frysk monitor for a process model event, such as a process exiting, that request is implemented by adding an observer (or monitor) to the objects to which the request applies. When the corresponding event occurs, the observers are notified.

4.2.2 Frysk's Language Model

Corresponding to the run-time state of a high-level program, Frysk provides a run-time language model. This model provides an abstraction of a running-program's stack (consisting of frames), variables and objects.



The model is then made available to the user through the source window's stack and source code browsers:



5 Future Direction

Going forward, Frysk's development is expected to be increasingly focused on large complex and distributed applications. Consequently Frysk is expected to continue pushing its available technology.

Internally, Frysk has already identified limitations of the current Linux Kernel debugging interfaces (`ptrace` and `proc`). For instance: that only the thread that did the attach be permitted to manipulate the debug target, or that waiting on kernel events still requires the juggling of `SIGCHLD` and `waitpid`. Addressing these issues will be critical to ensuring Frysk's scalability.

At the user level, Frysk will continue its exploration of interfaces that allow the user to analyze and debug increasingly large and distributed applications. For instance, Frysk's interface needs to be extended so that it is capable of visualizing and managing distributed applications involving hundreds or thousands of nodes.

6 Conclusion

Through the choice of a modern programming language, and the application of modern software design techniques, Frysk is well advanced in its goal of creating an intelligent, distributed, always-on monitoring and debugging tool.

7 Acknowledgments

Thanks goes to Michael Behm, Stan Cox, Adam Jocksch, Rick Moseley, Chris Moller, Phil Muldoon, Sami Wagiaalla, Elena Zannoni, and Wu Zhou, who provided feedback, code, and screenshots.

Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin