Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

Frysk 1, Kernel 0?

Andrew Cagney
Red Hat Canada, Inc.
cagney@redhat.com

Abstract

Frysk is a user-level, always-on, execution analysis and debugging tool designed to work on large applications running on current Linux kernels. Since Frysk, internally, makes aggressive use of the `utrace`, `ptrace`, and `/proc` interfaces, Frysk is often the first tool to identify regressions and problems in those areas. Consequently, Frysk, in addition to its GNOME application and command line utilities, includes a kernel regression test-suite as part of its installation.

This paper will examine Frysk's approach to testing, in particular the development and inclusion of unit tests directly targeted at kernel regressions. Examples will include a number of recently uncovered kernel bugs.

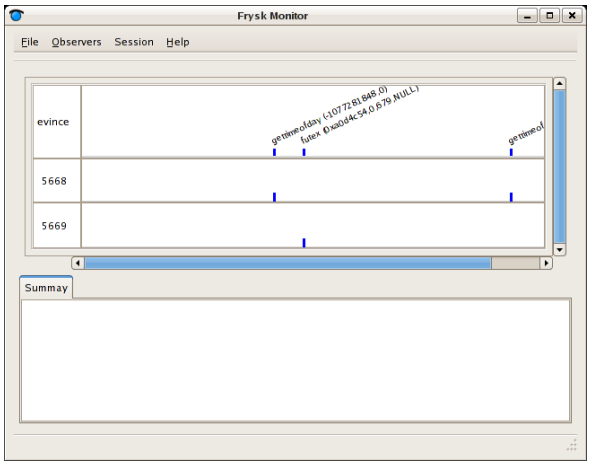

Figure 1: Frysk's Monitor Tool

1 Overview

This paper will first present an overview of the Frysk project, its goals, and the technical complexities or risks of such an effort.

The second section will examine in more detail one source of technical problems or risk—Linux's `ptrace` interface—and the way that the Frysk project has addressed that challenge.

In the concluding section, several specific examples of problems (both bugs and limitations) in the kernel that identified will be discussed.

2 The Frysk Project

The principal goal of the Frysk Project is to develop a suite of always-on, or very-long-running, tools that allow the developer and administrator to both monitor and debug complex modern user-land applications. Further, by exploiting the flexibility of the underlying Frysk architecture, Frysk also makes available a collection of more traditional debugging utilities.

In addition, as a secondary goal, the Frysk project is endeavoring to encourage the advancement of debug technologies, such as kernel-level monitoring and debugging interface (e.g., `utrace`), and debug libraries (e.g., `libdwfl` for DWARF debug info), available to Linux users and developers.

2.1 Frysk's Tool Set

A typical desktop or visual user of Frysk will use the monitoring tool to watch their system, perhaps focusing on a specific application (see Figure 1).

When a problem is noticed Frysk's more traditional debugging tool can be used to drill down to the root-cause (see Figure 2).

For more traditional command-line users, there are Frysk's utilities, which include:

- `fstack` – display a stack back-trace of either a running process or a core file;
- `fcore` – create a core file from a running program;

```
$ fcatch /usr/lib/frysk/funit-stackframe
fcatch: from PID 17430 TID 17430:
SIGSEGV detected - dumping stack trace for TID 17430
#0 0x0804835c in global_st_size () from: \ldots/funit-stackframe.S#50
#1 0x08048397 in main () from: \ldots/funit-stackframe.S#87
#2 0x00c2d4e4 in __libc_start_main ()
#3 0x080482d1 in _start ()
```

Figure 3: Running the `fcatch` utility

```
$ fstep -s 1 ls
[3299]  0xbfb840        mov    %esp,%eax
[3299]  0xbfb842        call   0xbfbf76
[3299]  0xbfbf76        push   %ebp
[3299]  0xbfbf77        mov    %esp,%ebp
[3299]  0xbfbf79        push   %edi
[3299]  0xbfbf7a        push   %esi
....
```

Figure 4: Running the `fstep` utility

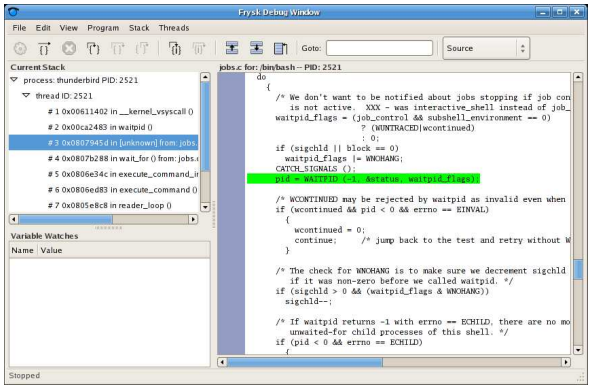

Figure 2: Frysk's Debugger Tool

- `fstep` – instruction-step trace a running program (see Figure 4);
- `fcatch` – catch a program as it is crashing, and display a stack back-trace of the errant thread (see Figure 3);

and of course the very traditional:

- `fhpd` – command-line debugger.

3 The Frysk Architecture

Internally, Frysk's architecture consists of three key components:

- kernel interface – handles the reception of system events (e.g., clone, fork, exec) reported to Frysk by the kernel; currently implemented using `ptrace`.
- the core – the engine that uses the events supplied by the kernel to implement an abstract model of the system.
- utilities and GNOME interface – implemented using the core.

4 Project Risks

From the outset, a number of risks were identified with the Frysk project. Beyond the technical complexities of building a monitoring and debugging tool-set, the Frysk project has additional "upstream" dependencies that could potentially impact on the project's effort:

- `gcj` – the Free GNU Java compiler; Frysk chose to use `gcj` as its principal compiler.
- `Java-GNOME` bindings – used to implement a true GNOME interface; Frysk is recognized as an early adopter of the Java-GNOME bindings.
- Kernel's `ptrace` and `/proc` interfaces – currently used by Frysk to obtain system information;

it was recognized that Frysk would be far more aggressive in its use of these interfaces than any existing clients and hence was likely to uncover new problems and limitations.

- Kernel's `utrace` interface – the maturing framework being developed to both replace and extend `ptrace`

The next two sections will review the Frysk–Kernel interface, problems that were encountered, and the actions taken to address those problems.

5 Managing Project Stability—Testing

One key aspect of a successful project is its overall stability, to that end the quality of testing is a key factor. This section will identify development processes that can both help and hinder that goal, with specific reference to the kernel and its `ptrace` and `/proc` interfaces.

5.1 Rapid Feedback—Linux Kernel

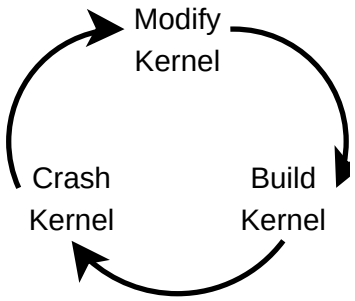

Figure 5: Short Feedback Cycle—Kernel

Linux's rapid progress is very much attributed to its open-source development model. In addition, and as illustrated by Figure 5, the Linux developer and the project's overall stability also benefits from very short testing and release cycles—just the act of booting the modified kernel is a significant testing step. Only occasionally do problems escape detection in the initial testing and review phases.

5.2 Slow Feedback—`ptrace` Component Clients

In contrast, as illustrated by Figure 6, the short development and testing cycle fails when a component is only occasionally exposed to use by its clients. The long lead-in time and the great number of changes that occur before an under-used kernel component is significantly tested greatly increases the risk that the kernel component will contain latent problems.

The `ptrace` interface provides a good illustration of this problem. Both changes directly to that module, such as a new implementation like `utrace`, or indirectly, such as modifications to the `exec` code, can lead to latent problems or limitations that are only detected much later when the significantly modified kernel is deployed by a distribution. Specific examples of this are discussed further in the third section.

5.3 Quickening the Feedback

Frysk, being heavily dependent on both the existing `ptrace` interface and the new `utrace` interface, recognized its exposure very early on in its development. To mitigate the risk of that exposure, Frysk implemented automated testing at three levels:

1. Integration test (Frysk) – using Dogtail (a test framework for GUI applications) and ExpUnit (an `expect` framework integrated into `JUnit`), test the user-visible functionality of Frysk.

2. Unit test (Frysk) – applying test-driven development ensured the rapid creation of automated tests that exercised Frysk's internal interfaces and external functionality; the unit tests allow Frysk developers to quickly isolate a new problem down to a sub-system and then, possibly, that sub-system's interaction with the kernel.

3. Regression test ("upstream") – where the root cause of a problem was determined to be an "upstream" or system component on which Frysk depended (e.g., kernel, compiler, library), a standalone automated test demonstrating the specific upstream problem was implemented.

Further, to encourage the use of these test suites, and ensure their likely use by even kernel developers, these test suites were included in the standard Frysk installation (e.g., in Fedora the `frysk-devel` RPM contains all of Frysk's test suites).

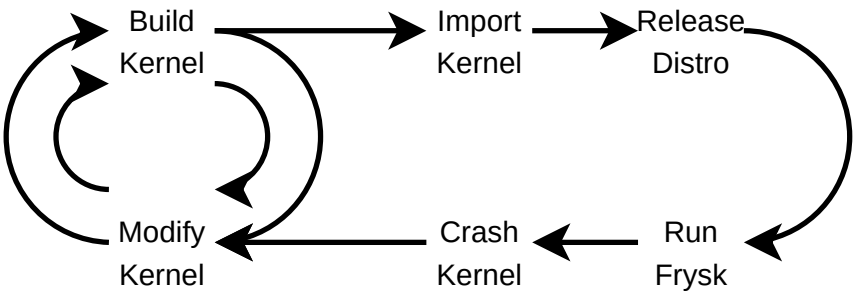

Figure 6: Long Feedback Cycle—`ptrace` interface

```
$ /usr/lib/frysk/fsystest
SKIP: frysk2595/ptrace_after_forked_thread_exits
PASS: frysk2595/ptrace_after_exec
....
PASS: frysk3525/exit47
PASS: frysk3595/detach-multi-thread
PASS: frysk2130/strace-clone-exec.sh
XFAIL: frysk2595/ptrace_peek_wrong_thread
```

Figure 7: Running Frysk's `fsystest`

Figure 7 illustrates the running of Frysk's "upstream" or system test suite using `fsystest`, and Figure 8 illustrates the running of Frysk's own test suite, implemented using the `JUnit` test framework.

6 Problems Identified by Frysk

In this final section, two specific tests included in Frysk's "upstream" test suite will be described.

6.1 Case 1: Clone-Exec Crash

In applying test-driven development, Frysk developers first enumerate, and then implement all identifiable sequences of a given scenario. For instance, to ensure that `exec` can be correctly handled, the Frysk test suite sequences many scenarios including the following:

- 32-bit program exec'ing a 64-bit program
- main thread exec'ing
- non-main thread exec'ing

The last case, as illustrated in Figure 9, has proven to be especially interesting. When first implemented,

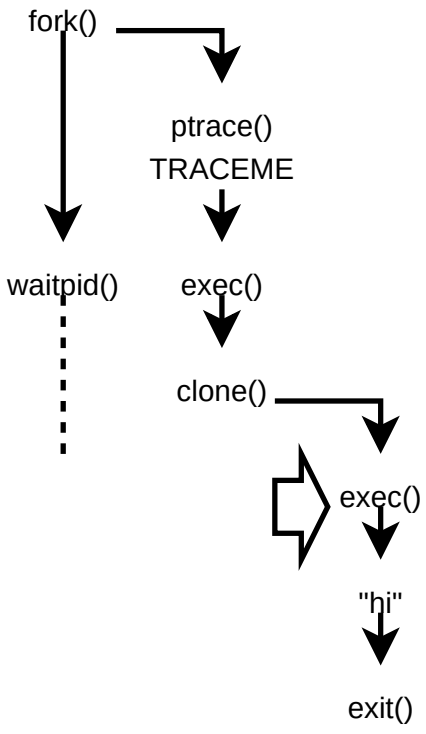

Figure 9: Clone Exec Crash

```
$ /usr/lib/frysk/funit
Running testAssertEOF(frysk.expunit.TestExpect) ...PASS
Running testTimeout(frysk.expunit.TestExpect) ...PASS
Running testEquals(frysk.expunit.TestExpect) ...PASS
Running testIA32(frysk.sys.proc.TestAuxv) ...PASS
Running testAMD64(frysk.sys.proc.TestAuxv) ...PASS
Running testIA64(frysk.sys.proc.TestAuxv) ...PASS
Running testPPC32(frysk.sys.proc.TestAuxv) ...PASS
Running testPPC64(frysk.sys.proc.TestAuxv) ...PASS
....
```

Figure 8: Running Frysk's `funit`

it was found that the 2.6.14 kernel had a regression causing Frysk's unit-test to fail—the traced program would core dump. Consequently, a standalone test `strace-clone-exec.sh` was added to Frysk's "upstream" test suite demonstrating the problem, and the fix was pushed upstream.

Then later with the availability of the 2.6.18 kernels with the `utrace` patch, it was found that running Frysk's test suite could trigger a kernel panic. This was quickly isolated down to the same `strace-clone-exec.sh` test, but this time running the test caused a kernel panic. Since the test was already widely available, a fix could soon be written and incorporated upstream.

6.2 Case 2: Threaded ptrace Calls

Graphical debugging applications, such as Frysk, are often built around two or more threads:

- an event-loop thread handling process start, stop, and other events being reported by the kernel.
- a user-interface thread that responds to user requests such as displaying the contents of a stopped process' memory, while at the same time ensuring that the graphical display remains responsive.

As illustrated in Figure 10, Linux restricts all `ptrace` requests to the thread that made the initial `PTRACE_ATTACH`. Consequently, any application using `ptrace` is forced to route all calls through a dedicated thread. In the case of Frysk, initially a dedicated `ptrace` thread was created, but later that thread's functionality was folded into the event-loop thread.

The "upstream" test `ptrace_peek_wrong_thread` was added to illustrate this kernel limitation.

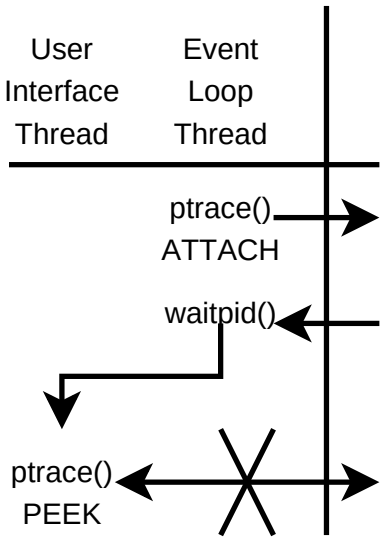

Figure 10: Multi-threaded ptrace

7 Conclusion

As illustrated by examples such as the Exec Crash bug described in Section 6.1, Frysk, by both implementing an "upstream" test suite (focused largely on the kernel) and including that test suite in a standard install, has helped to significantly reduce the lag between a kernel change affecting a debug-interface on which it depends (such as `ptrace`) and that change been detected and resolved.

And the final score? Since Frysk's stand-alone test suite is identifying limitations and problems in the existing `ptrace` and `/proc` interfaces and the problems in the new `utrace` code, this one can be called a draw.

8 Acknowledgments

Special thanks to the Toronto Windsurfing Club Hatteras Crew for allowing me to write this, and not forcing me to sail.

References

The Frysk Project, `http://sourceware.org/frysk`

Clone-exec Crash Bug, `http://sourceware.org/bugzilla/show_bug.cgi?id=2130`

Threaded ptrace Calls Bug, `http://sourceware.org/bugzilla/show_bug.cgi?id=2595`

Roland McGrath, *`utrace` Patches*, `http://people.redhat.com/roland/utrace/`

The JUnit Project, `http://www.junit.org/index.htm`

The Java-GNOME Project, `http://java-gnome.sourceforge.net/`

GCJ, `http://gcc.gnu.org/java/`

ExpUnit, `http://sourceware.org/frysk/javadoc/public/frysk/expunit/package-summary.html`

Dogtail, `http://people.redhat.com/zcerza/dogtail/`

`expect`, `http://expect.nist.gov/`