Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada

NPTL Stabilization Project

NPTL Tests and Trace

Sébastien DECUGIS
Bull S.A.
sebastien.decugis@bull.net

Tony REIX
Bull S.A.
tony.reix@bull.net

Abstract

Our project is a stabilization effort on the GNU libc thread library NPTL—Native POSIX Threading Library. To achieve this, we focused our work on extending the pool of open-source tests and on providing a tool for tracing the internal mechanisms of the library.

This paper introduces our work with a short status on test coverage of NPTL at the beginning of the project (February 2004). It explains how we built the prioritized list of NPTL routines to be tested. It then describes our methodology for designing tests in the following areas: conformance to POSIX standard, scalability, and stress. It also explains how we have simplified the use of the tests and the analysis of the results. Finally, it provides figures about our results, and it shows how NPTL has evolved during year 2004.

The paper goes on to explain how this NPTL Trace Tool can help NPTL users, and hackers, to understand and fix problems. It describes the features of the tool and presents our chosen architecture. Finally, it shows the current status of the project and the possible future extensions.

1 Introduction

NPTL library was first released on September 2002 and merged with the glibc about sixteen months later. It was meant from the beginning to replace the LinuxThreads implementation, and therefore become the standard thread library in GNU systems. The new library provides full conformance to the POSIX[1] requirements, including signal support, very good performance and scalability.

Porting from LinuxThreads to NPTL was intended to be transparent; however, there are several cases where software using NPTL must be modified. There are some documented changes, such as signal handling or `getpid()` behavior. There are also changes in the application dynamics, such as those caused by threads being created more quickly. A user application coded with incorrect assumptions about multi-threaded programming can fail because of some of the semantic changes; such problems are very difficult to debug. We had the opportunity to work with IBM on some of their internal *BugZilla* reports, and in many cases the problem appeared because of changes in appli-

[1]The POSIX® standard refers to the IEEE Std 1003.1, a.k.a. Single UNIX Specification [1] v3. The current version is the 2004 Edition and includes Technical Corrigendum 1 and 2. POSIX is a registered trademark of the IEEE, Inc.

cation dynamics. Last but not least, NPTL is still under development. New features are being added from time to time. Fixes and optimizations are also frequent. All these code modifications have the potential to introduce new bugs.

Before NPTL could be used reliably in complex applications on production systems, it needed more substantial testing and validation. Any production system providing reliable applications should not crash or hang simply because the threading library is not stable. On the other hand, the new library provides very good performance and therefore is of great interest for these same systems.

To continuously improve the stability and quality of NPTL as it evolves, as well as to shorten the stabilization period after each change, we developed a robust set of regression and stress tests. Ideally, these tests would be run frequently during NPTL development to look for regressions and the tests can be augmented as new features are added. These tests should cover as many APIs, arguments to the APIs, and threading semantics as possible. The tests must remain independent of the implementation of the threading library so that the tests will not need to be changed each time the implementation changes. We will see in the next chapter how were specified and developed a list of tests, how we tried to make these tests runs as simple and user-friendly as possible, and finally we will show NPTL evolution, from the test results point of view, through year 2004.

As we have seen previously, many of the problems developers have to face when they port an application from LinuxThreads to NPTL are due to bugs located in their application, not in NPTL. Bugs dealing with multi-threading are particularly difficult to isolate and reproduce most of the time. As an example, when you run the program step-by-step in a debugger, the thread creation time is totally different than when it runs outside the debugger. These bugs can also depend on the machine load, on a device access slowing only one of the threads, or a multitude of factors, resulting in weeks of research and testing for an application developer. Moreover, many POSIX standard interfaces are quite intricate, and many programmers do not test all return codes from NPTL routines. At best, an application which receives an unexpected error code may crash; at worst, the application may corrupt data silently.

To solve these issues, we have developed a trace tool for NPTL, called POSIX Threads Trace Tool (PTT). This tool keeps track of all NPTL related events, such as thread creation, lock acquisition, with little impact on the application. By tracing the library internals, we can understand the chain of events which lead to a hang or strange behavior in the application. We can also understand how the application is really using NPTL functions, measure lock contention, and optimize both the NPTL implementation and the application's use of NPTL. Finally, these traces can prove that a bug is in NPTL or in the kernel, rather than in the application. The third chapter of this paper is dedicated to this trace tool. It attempts to show the limitations of existing tools, then describes the features of our tool and how these features can be used efficiently to solve real situations. It also shows the tool internals and its current limitations and future directions.

The paper concludes with an overview of the remaining work to do on NPTL, NPTL tests, and NPTL trace tool, in order to obtain a production quality level in this open-source product. It shows the current use of the tests in the library and kernel development process. It also shows that this testing effort is necessarily not a "one-shot" project, and that more people should be involved in projects like this one. As for the trace, it shows how the trace tool can be extended into a dynamic code checker, or into a profiling tool, with a minimal effort. It also deals with how this modified NPTL can be set up in a production environment, and why people should use this tool.

2 NPTL Tests

The first part of our project consisted of improving the test coverage for the NPTL library. Our goal was to be as exhaustive as possible, at least as far as POSIX requirements are concerned. We focused on the POSIX standard [1] among all standards the NPTL is supposed to conform to, because it is largely used on other platforms, and so is important for ensuring portability of an application, and because reference is made in the library name—Native *POSIX* Thread Library—which means it is the first standard one would expect NPTL to conform to.

2.1 Situation on March 2004

When we started our project in early 2004, we isolated three open-source projects which provided test cases for NPTL.

The first one is the *GNU lib C* project [glibc] itself. NPTL source tree contains test cases that can be run against the freshly compiled glibc by issuing the `make check` command. These tests—about 160 files at that time—are not documented at all and hardly commented. Their naming convention is the only hint to guess what each test is supposed to do. We had a hard time reading each test case and writing a short abstract on what the test is really doing. As a synthesis, these tests are mostly regression tests which test for very specific features, and test coverage for each library routine is far from adequate or complete. Moreover, the tests are often very close to NPTL internals, which means more maintenance when the library implementation changes. These tests are useful for the glibc developers, but are by design too closely linked to testing implementation specifics to be usable as a proof of reliability or indicator of conformance.

The second project we focused on is the *Open POSIX Test Suite* [OPTS]. This is a pure test project, with a lite harness—the only constraint on a test case is its return value—and a simple structure, at least for the regression tests. For each library routine, an XML file contains a set of assertions that describe the POSIX standard requirements for this routine, and then the test cases are named according to the assertion they are testing. Extracting the coverage information is quite straightforward from this structure. The test cases are also often well documented, with few exceptions where the comments do not match the content.

The third project we considered is the *Linux Test Project* [LTP]. This is the most used open-source test project for Linux, but it appeared that it provides very few test cases for NPTL, aside from those of the OPTS which is included. Moreover, the structure is more complex and the format for test cases is more rigid than in the OPTS.

After this analysis, we decided to release our test cases to the OPTS, as they would later be included in LTP with the complete OPTS new release. In situations where we would have to write implementation-dependent test cases, they would be submitted to the glibc project directly, but we did our best to avoid NPTL-internals dependent code, as it would require more maintenance.

2.2 Prioritized list

Our next step was to find what to test. NPTL contains more than 150 routines, so we had to establish our priority list based on the following criteria:

1. functions which are used the most frequently;
2. functions which are complex enough to possibly contain bugs, based on their algorithm; and
3. functions which are not just a wrapper to the kernel—as we are not testing the kernel.

To find out which functions are the most used, we chose seven multi-threaded applications representative of several computer science domains where multi-threading is frequently used. The selected software were: two different *Java Virtual Machine*s; *JOnAS*, an open-source Java application server, compiled with *gcj*; the *Apache* web server; the *squid* web cache and proxy; the *MySQL* database server; and *GLucas*, a scientific software described in the next chapter. Each application was analyzed with the *nm* utility to find out which NPTL routines were used. We also included a personal opinion based on our past experience with each routine, to establish the list.

This work has resulted in a complete list of functions split into 4 groups, from the most important to test to the less important. The first group (most important) contains 15 functions, dealing with *threads*, *mutexes* and *condvars*. The second group contains 27 functions, dealing with *threads*, *signals*, *cancellation* and *semaphores*. The complete list is available on our website [2]. The remaining functions belong to groups three and four. Even if NPTL contains 150+ functions, many of those functions are only used to change a value in a structure (attribute), so the bug probability is really small. With groups 1, 2 and 3 we cover almost all the functions which can encounter problems. At this time, only groups 1 and 2 have been completely tested. There is still a great amount of work remaining to complete the test coverage—this will be detailed later in this paper.

2.3 Methodology

We had to design a method for test writing. We based it on the OPTS method.

For each library routine to test, the first step was to analyze the POSIX standard and extract each assertion that the function has to verify to be compliant. For some functions the standard appeared to be unclear or contradictory. In these cases, we opened requests for clarification in the Austin Revision Group [3], so that the next Technical Corrigendum for the standard would clarify the obscure parts.

In the next step, these assertions were compared to those already present in the OPTS, and the *assertions.xml* file was updated according to the differences we found. Most of the differences we encountered so far were due to the POSIX standard evolution since the OPTS was first released.

The third step in the design was to check each existing test case for a given assertion, find out possible errors, try to check that all situations were tested, and list the missing cases which had to be written. For some assertions, we also had to specify stress tests to be written in order to be exhaustive, or when we could not figure another way to test a particular feature. We also specified scalability tests to be written for some functions where scalability is important, even if this is more a quality of implementation issue than part of the POSIX standard.

At each step of this process, for each function, we posted an article in our project forum, publicly available and accessible from our website

[2]. This allowed other people to check how they could help or see the rationale for a particular test.

Once our design was complete, we had to write the test cases and submit them to the OPTS project. We wrote three kinds of test cases: *conformance*, *stress* and *scalability*.

A conformance test runs for a short period of time and returns a value representing its result: PASSED, FAILED, UNRESOLVED, etc. See the OPTS documentation for a detailed explanation of return codes.

A stress test runs forever until it is interrupted with SIGUSR1 (means success) or a problem occurs (means failure). Most of the stress tests are very resource-consuming and are meant to be run alone in the system. In this way, it is possible to identify the cause of a failure, when any occurs.

A scalability test loops on a given operation until the number of iterations is reached or until failure, and saves the duration of each iteration. Then, measures are parsed with a mathematical algorithm which tells if the function is scalable (constant duration) or not (duration depends on the changing parameter). The algorithm is based on the least squares method to model the results. The table of measures can also be output and used to generate a graph of the results with the *gnuplot* tool. Figure 1 gives an example of such a graphical output. It shows the duration of `sem_open()` and `sem_close()` operations with an increasing number of opened semaphores in the system. Other examples can be found in our forum.

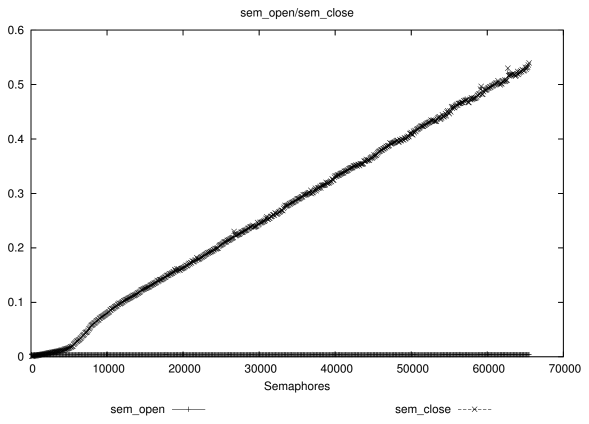

Figure 1: Graphical output sample

2.4 TSLogParser tool

To be useful, the tests must be run frequently, and the results must be easy to analyze and compare with other runs (references). Whereas running the tests is quite straightforward with OPTS (just set up the flags and run `make`), the analysis can be a real pain. As an example, after we completed the test writing for our first group of functions, we ran a complete test campaign on several Linux distributions with several hardware architectures—i686, PowerPC, ia64. We got a total of nine different configurations, and three runs on each configuration, which resulted in a total of about 50,000 test case results to digest. Needless to say, we needed automated tools to extract the useful information!

In many cases, comparing several runs and finding quickly what the differences are in detail is all we need. That supposes we have an arbitrary reference, to compare new code results to. But comparing huge log files is far from being easy. Using the *diff* tool is not a solution, as there are expected differences between the runs—order of test case execution, timestamps, random values— and a long time can be spent doing the analysis.

Another approach is to first make a synthesis of each run, and then only compare the synthesis. This is quite easy to achieve with tools such as *grep* and *wc*. The Scalable Test Platform (discussed in the next section), for example, uses this kind of summary tool. Anyway, this approach has some drawbacks. When a new failure appears, it is not possible to find out which test is failing. Also, if the success/failure distribution remains constant, while not involving the same individual tests, you won't see anything with your tool.

To address all these issues, we have designed a new tool: *TSLogParser* [4]. The main idea is to parse the log file of a test suite run and save the results and detailed information about each test into a database; and then be able to access all this information through a web interface. It allows filtering of results, to show only partial information or to access all details in just a few clicks. It also makes comparing several runs quite easy.

The structure of this tool has been designed to allow several kinds of test suites to be parsed and displayed the same way. The parser module which saves the log file into the database is written as a plug-in. The visualization and administration interfaces are not dependent on the test suite format. The current implementation is written in PHP and has been used with Apache and MySQL. It is able to compare up to 10 OPTS runs at once on a standard workstation. It also extracts statistical information from each run and allows filtering according to test status—for example one may want to hide all the successful tests or show only tests that end with a segmentation fault.

This tool has made the analysis of OPTS run results a fast and easy operation. It is a must-have—in our opinion—for anyone who is using the OPTS.

2.5 Scalable Test Platform

Another frequent issue in testing is that the active developers often lack the resources—time, hardware—to run complete test campaigns frequently. This can be solved, thanks to the *Scalable Test Platform* [STP] and *Patch Lifecycle Manager* [PLM] projects from *Open Source Development Labs* (OSDL).

PLM tracks the official kernel patches and allows uploading of new patches (either manually or automatically). STP allows people to request runs against tests, against any PLM patch, with a choice of Linux distributions and machine hardware. Our project contributed to STP by making the OPTS runnable through its interface. There is also a work in progress to bring the same patch feature that PLM provides for the glibc of the test system.

Once the requested test run completes, an email is sent to the requester with a summary of the results, and the complete log file is available for download. There is another work in progress to make the results available through the TSLog-Parser interface, because as we already discussed a summary can sometimes not contain enough information.

A very interesting feature of the PLM project is the ability to automatically pull new kernel patches and run a bunch of tests in STP—including the OPTS—against the new patched kernel. This allows very quick detection when new problems appear.

2.6 Situation on March 2005

After sixteen months of active work on this project, we are reaching the end of our credits. During this period, we were able to analyze and write test cases for all our 42 most important NPTL functions. A total of 246 conformance tests, 9 scalability tests and 16 stress tests have been written. These 42 functions correspond to 283 distinct assertions in POSIX, of which 246 (90%) are now covered by the OPTS and

135 (55% of OPTS) were contributed within our project. Figure 2 shows the evolution of the number of test cases (upper plot) and functions tested (lower plot) during our project. It only refers to our contribution, not to the complete OPTS project. The horizontal step during November 2004 corresponds to our first test campaign. The vertical step on February 2005 is due to semi-automated test generation for some signal-related functions. It is interesting to note that both plots are almost identical. This means that the amount of work required for each function to complete the OPTS work is almost the same for all functions. This needed work can be due to POSIX evolutions, as well as incomplete or invalid OPTS test cases.

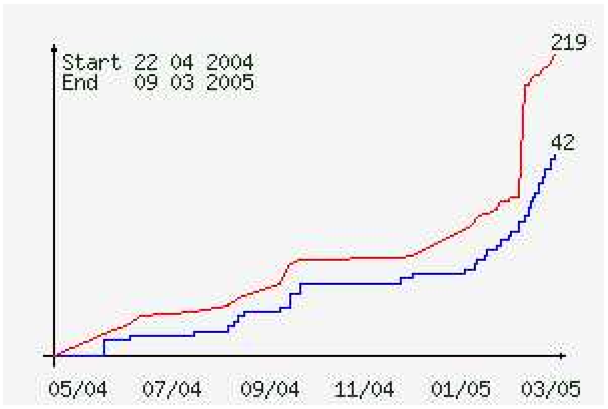

Figure 2: Project progression

Thanks to these test cases, a total of 22 defect reports have been issued—21 in the glibc and 1 in the kernel—most of which have been fixed in recent releases. The kernel defect deals with the scheduler and the `SCHED_RR` policy behavior on SMP machines. The glibc defects are either conformance bugs (wrong error code returned, bad `#include` files or symbol requirements), or functional bugs (flags role in `sigaction()`, behavior of timeouts with condvars), or else just bugs (segmentation faults, hangs, unexpected behaviors). We've also found a scalability issue with the function `sem_close()`, the duration of which depends on the number of opened semaphores.

In the meantime, 5 enhancement requests have been issued to the Austin Revision Group, about obscure or incomplete points in the POSIX standard. These requests addressed issues in `pthread_mutex_lock()`, `pthread_cond_wait()`, `pthread_cond_timedwait()`, `sigaction()`, and `sem_open()`. All have been accepted or are still pending.

The most important part of our project is not the number of bugs we have found, but the number of assertions which are now tested. For 42 functions we analyzed, almost *all* of what can be tested *is* now tested in OPTS. The test cases for these 42 functions cover all the current POSIX requirements.

2.7 NPTL Evolution over year 2004

As an example, we have run the current OPTS release with *Fedora Core 1* (FC1), *Fedora Core 2* (FC2) and *Fedora Core 3* (FC3) distributions, as well as an 'unstable' Fedora Core 3 update. After analyzing the results with the TSLogParser tool, we have come to find some interesting conclusions, detailed in the next paragraphs. This kind of analysis is very easy to achieve and can help tracking new bugs very quickly. Anyway, as the TSLogParser tool is interactive, we cannot reproduce its output in this document, and encourage the reader to check the tool web site [4] for examples, including the data discussed here.

AIO operations. Some test cases related to Asynchronous I/O operations, such as `aio_read()` or `aio_write()`, returned PASS with FC1 and FC2 and return FAIL or hang with the more recent distributions. This may indicate a bug in the new kernels or in the glibc.

Clock routines. Some tests related to the clock routines (`clock_settime()`, `nanosleep()`) did not pass in FC1, but things have been fixed since FC3.

Message queues. The message queues routines were not implemented in FC1, so the related tests reported a 'build failure' status. Everything is fine since FC2.

Sched routines. A few test cases related to the sched routines (`sched_setparam()`, `sched_setscheduler()`) won't compile in the latest FC3 update, whereas they passed in the previous releases.

There are some other test cases which would be worth a deeper investigation, but we won't enter into the details here. Reproducing these results is quite easy, and it would be valuable for Linux and the glibc that more people carry on this kind of work.

3 NPTL Trace

The second part of our work was dedicated to tracing NPTL.

3.1 Why Tracing?

Since more and more HyperThreaded or Multi-Core processors are available, it is expected that the design of many new applications will use multi-threading for running several tasks simultaneously and concurrently, in order to take profit of nearly all the available power of the machine.

Writing a portable multi-threaded application is a complex task: the POSIX Thread standard is not easy to understand. It provides ten kinds of objects: Thread, Mutex, Barrier, Conditional Variable, Semaphore, Spinlock, Timer, Read-Write lock, Message queues, and TLD (Thread Local Data). These objects are available under eighteen options: **BAR**, **CS**, **MSG**, **PS**, **RWL**, **SEM**, **SPI**, SS, **TCT**, **THR**, **TMO**, TPI, TPP, TPS, **TSA**, **TSH**, TSP, **TSS** [2] that may be supported or not by Operating Systems. (See [1] for the meaning of each option). NPTL provides about 150 different routines to manage the POSIX objects.

Also, "**anything can occur at any time**": a program must not assume that an Event A always occurs before—or after—Event B. That may be true on a small machine; but it will certainly be untrue some day on a bigger and faster machine at a customer site. That makes writing a multi-threaded application more complex than initially expected.

On Linux, NPTL is quite perfectly compliant with the POSIX Threads standard. Since several parts of the POSIX Threads standard are unspecified, they can be provided differently by two POSIX Threads libraries. So porting an application from another Operating System (though providing the same POSIX Threads objects and routines) to Linux may lead to bad surprises. Being able to quickly understand why an application behaves badly (hang, unexpected behavior, etc.) is critical for customers. Often, reproducing the problem in support labs is not possible since it may appear after days of computation. This may require sending a Linux guru to the customer site. Also, understanding quickly if the problem is in the application, in NPTL, or in the Linux Kernel is critical.

Analyzing a multi-threaded application showing a race condition or a hang with a debugger is not the right approach because it will certainly modify the way threads are scheduled, possibly causing the problem to disap-

[2]The options provided by recent GNU libc are highlighted in **bold**.

pear. The right approach is by using a less intrusive method, such as a trace tool. A NPTL trace tool enables recording of the most important multi-threading operations of an application or the main steps of NPTL with a minimal impact to the application (performances and flow of execution of threads). The trace can be analyzed once the problem has appeared and the application has stopped: it is **Post-Mortem Analysis**. If the impact on a critical application is acceptable, one can even continuously record the last few thousand traces so that analyzing a failure can be done when it occurs for the first time: it is **First Failure Data Capture**.

Why not use Linux Trace Toolkit [LTT]? First, LTT is designed to trace events in the kernel and not to trace programs in the user space. Second, LTT uses functions (like `write()`) that cannot be used when tracing NPTL. (See section 3.3.1 on page 120).

Why not simply use some *wrapper* enabling trace of only the calls of the application to NPTL routines? Because such a tool does not enable to analyze both the behavior of NPTL and that of the application. And, since it also requires to put in place a complex mechanism for collecting and storing traces, it is worth also tracing the behavior of NPTL routines, by adding traces inside its code.

So, as explained hereafter, we finally decided to design our own NPTL tracing tool.

3.2 Goals

At the beginning of 2004, when we started to add new tests for NPTL, we also started to study a NPTL trace tool. After discussing the design of such a tool with people involved in thread technology and in the glibc (IBM: F. Levine, E. Farchi; HP: J. Harrow; Intel: I. Perez-Gonzalez; etc.), we decided to propose to students from French Universities to study the architecture of the tool and to build it.

The POSIX Threads NPTL Trace Tool **[PTT]** has been designed to provide a solution to the requirements discussed previously. It addresses the three kinds of users described hereafter.

3.2.1 Users

We have studied the needs of three different kinds of users:

A **developer** in charge of writing, porting or maintaining a multi-threaded application. He mainly needs to see when his program calls NPTL routines and when it exits from them, with details about the parameters. He wants to be able to easily and quickly switch from a fast untraced NPTL to a traced NPTL, and vice-versa, without recompiling his application. When using the traced NPTL, the maximum acceptable decrease in the performances of his application is 10%.

A member of a **support team** that provides Linux skills to other people who write, test or use applications. This kind of user has skills about the Linux kernel and the GNU libc and he needs to see what is happening inside NPTL. Also he is very interested in generating traces at customer sites and to analyze them in his own offices.

A **hacker of NPTL**. Since analyzing why NPTL does not perform as expected is not an easy task, it is crucial to provide help. This way, more people could contribute to analyzing the behavior of NPTL and fix problems.

3.2.2 Features

Using PTT is a four step process:

1. build or get a traced NPTL library;
2. trace the application and build a binary trace;
3. translate the binary file into a text file that will be parsed by another program or that will be read manually;
4. analyze the trace, possibly with a tool helping to handle many objects and traces.

Several features are required:

- do not break the POSIX conformance rules (mainly cancellation).
- enable several people to trace different applications at the same time.
- handle large volumes of traces due to an application running days and weeks before the problem occurs: keep only last traces or manage very large trace files.
- give meaningful names to NPTL objects rather than hexadecimal addresses, since the application may create hundreds or thousands of objects of each kind.
- dynamically switch from a light trace to a richer or full trace.
- filter the decoded trace based on various criteria (name or kind of object, etc).
- start/stop the trace while the application is running, and provide solutions for handling incomplete traces.
- handle applications that fork new processes that must be traced.
- handle bad situations (hang, crash, kill).

3.3 Architecture

The main idea is to handle a buffer in shared memory: the threads of the application write the traces in the buffer, while a daemon (launched as a separate process) concurrently and periodically reads the traces in the buffer and writes them into the binary file. Traces are concurrently added by the threads into the buffer at the time the events occur. Figure 3 provides a simplified description of the architecture of PTT.

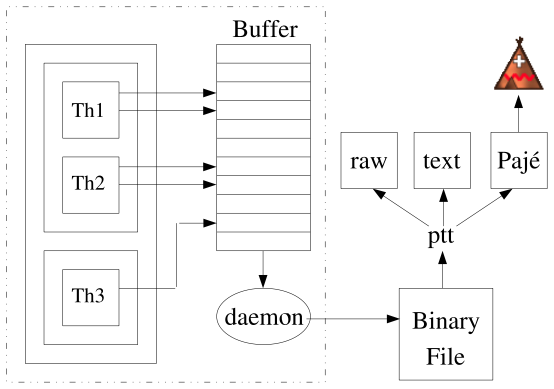

Figure 3: Architecture

3.3.1 POSIX Constraints

The architecture must take into account the following constraints.

- First, the **POSIX Threads standard** defines which routines can be a *Cancellation Point* (CP)[3]. POSIX defines three categories: the routines that shall be a CP,

[3]A Cancellation Point is a place where a thread can be canceled by means of `pthread_cancel()`. Such places appear when the cancellation *state* is set to *enabled*, and *type* is *deferred*.

those that cannot, and those that are undefined (free). It means that adding a CP into a routine that cannot have a CP is forbidden: routines like `printf()` cannot be called by trace code from inside NPTL routines. In few words, a NPTL trace mechanism can almost only write into memory !

- Second, tracing an application must have a minimum ***impact***. It means that the application must not run significantly slower and must not behave very differently than without the trace: the application must produce the same results and its threads should continue executing in the same order so that problems do not disappear.

3.3.2 Components

Events are written into a buffer. Then a daemon copies them to a binary file.

The basic component of the trace is an **event**. An event shows either a change in an attribute (state, owner, value, ...) of a NPTL object, or the calls (in / out) to any NPTL routine by the application. Sixty events have been defined for the four objects: Threads, Mutex, Barrier, CondVar. About 200 different events are expected to be defined when all routines are traced. As an example, eleven events have been defined for the Thread object: `THREAD_JOIN, _DETACH, _STATE_DEAD, _STATE_WAIT, _STATE_WAKE, _INIT, _CREATE_IN, _CREATE_OUT, _JOIN_IN, _JOIN_OUT, _SET_PD`. Each event is recorded in the buffer with useful data: time-stamp (for computing the elapsed time between two events), process Id, thread Id, and parameters. Events contain various amounts and kinds of data.

Adjacent events are grouped as a **trace point** in order to reduce the impact of the trace mechanism: only one call is done instead of two or more.

A circular **buffer** allocated in shared memory is used for storing the traces. If the buffer is not appropriately sized (too small for a given number of threads and processors), there is a risk of overflow: new traces are written over the oldest traces that the daemon is attempting to copy to the binary file. Buffer overflow is managed and produces a clear message. But its probability is nearly null, as explained hereafter.

A **daemon** is in charge of continuously monitoring the filling rate of the buffer. When a threshold is crossed, the daemon copies the traces to the binary file. One instance of the daemon is launched per application and behaves as the parent process of the application process.

One **binary file** is filled with traces for each traced application. It can be converted to text by means of a decoding tool. And its size can be greater than 2 Gigabytes.

3.3.3 Managing the Buffer

Correctly and efficiently managing the trace buffer was a quite complex task. Since using NPTL objects and routines (mutex) is forbidden, we used the atomic macros provided by the glibc.

We considered several solutions for managing the trace buffer:

- use two buffers: when one is full the buffers are switched and the threads write traces in the other one, enabling the daemon to save the traces to file without blocking the application threads, but with the risk of loosing traces.

- the same, but with blocking the threads and with no risk of loosing traces.
- use one buffer per processor in order to reduce the contention between traces.
- use one buffer per thread, suppressing all contention.
- use one buffer per process launched by the command to be traced.
- use one buffer for all processors, all processes and all threads launched by the command to be traced.

Each of these solutions has drawbacks and benefits about complexity, reliability and performance. We started looking in detail at the last solution. It appeared to be reliable, efficient, and not too complex, based on experiments we made on bi- and quad-processor machines.

The solution is based on the following two mechanisms:

1) When a thread needs to store trace data into the buffer, it first *reserves* the appropriate amount of space by increasing the *reserved* pointer in one **atomic** operation. Then it writes the trace data in the reserved space. And finally it increases the *written* pointer with the amount of written bytes by means of another **atomic** operation. With this approach, the buffer is never locked when threads reserve space and write traces.

2) The daemon continuously monitors the percentage of buffer already filled with traces. When the daemon decides that it is time to save the filled and reserved parts of the buffer, the daemon blocks all threads attempting to reserve more space in the buffer. Once all threads have completed writing events in the buffer (when *written* has reached *reserved*), the daemon releases the threads which restart reserving space in the buffer. Then the daemon writes the filled part of the buffer into the binary file. The goal is not to lose traces.

The figure 4 explains the main steps:

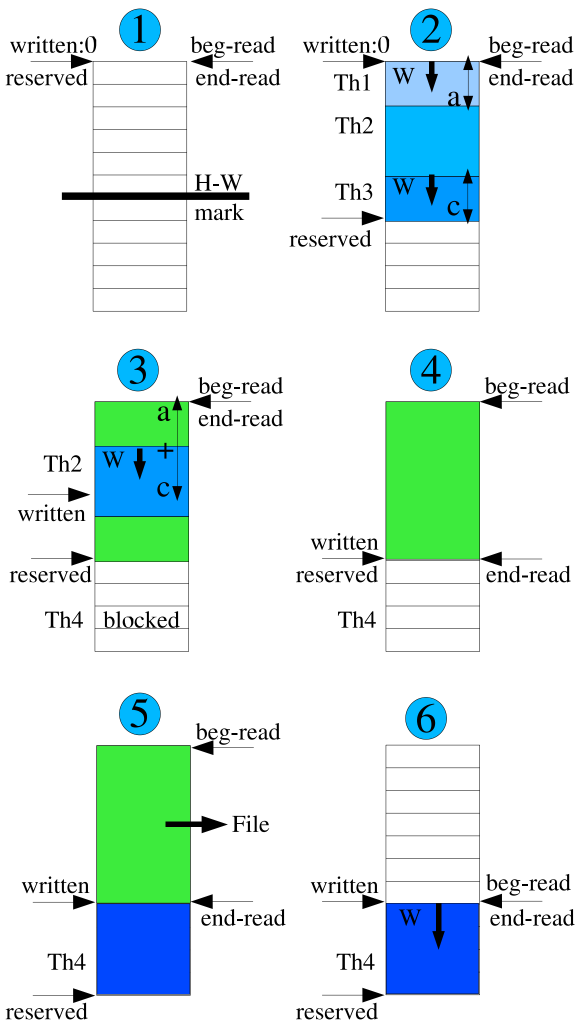

Figure 4: Buffer management

1) Start: No space has been reserved.

2) Threads 1, 2 and 3 have successively reserved the space they need for writing their trace. The reserved space has crossed the High-Water mark: the daemon now blocks the other threads attempting to reserve space. Threads

1 and 3 have started writing the trace data whereas thread 2 has not started yet.

3) Threads 1 and 3 have finished writing: an amount of *a*+*c* bytes of data has already been written. Thread 2 has started writing. Thread 4 is blocked.

4) Thread 2 has finished writing. Now the written space (*a*+*c*+*b*) is equal to the reserved space. The daemon knows which area must be saved to disk: Thread 4 is released.

5) The daemon is writing the trace data to the binary file. Thread 4 has reserved the needed space.

6) The daemon has finished writing the trace data. Thread 4 is writing its trace data.

An overflow may occur when threads write data in the buffer faster than the daemon empties it. Experiments have shown that it may appear only if the buffer is very small (let's say: 1 MB for one fast processor) and if the application is continuously writing traces due to many competing threads. Using a larger buffer is a good solution. By default, the threshold (indicating when it is time to empty the buffer) is set to half the size of the buffer. The size of the buffer for small and medium machines is computed as: $(MemSize * NberOfProcessors)/K$ where $K = 128$ by default. Thus, with a 1GB machine with 2 processors, the size of the buffer is 16 MB. We monitored the maximum usage of the buffer with various applications and the conclusion is that even an unrealistic application designed for writing PTT traces as fast as possible cannot overflow the buffer when K is 64. The applications we used never fill the buffer more than 60% before the daemon empties it. If needed, the user is able to use a more adequate buffer size, as a parameter given to the PTT launcher `ptt-view`.

If an overflow occurs, the threads of the application hang. After a time-out, the application is stopped by the daemon.

Other problems may also occur when a thread is canceled, hangs or dies.

- A thread can be canceled by means of the `pthread_cancel()` routine. The POSIX standard defines that an application can switch from and to two different cancellation modes: asynchronous or deferred (synchronous). In asynchronous mode, the thread can be canceled anywhere (if the cancellation state is *enabled*). In deferred mode, the thread can only be canceled in Cancellation Points.

  In order to guarantee that the trace data written in the buffer are always complete, the execution of the PTT trace mechanism is done in deferred mode (the previous cancellation *mode* is stored and then restored).

- A hang of the application can lead to 2 different cases. If an application thread hangs after it has reserved space in the trace buffer and before it has written its trace data, the daemon saves the last traces after waiting a time-out. If a thread hangs elsewhere, one must kill the application.

- When a thread runs into a Segmentation Fault or receives a kill signal, the daemon is warned and saves the last unsaved traces.

Moreover—as expected—once the application has completed its task and has returned, the daemon saves the last unsaved traces.

In order to simplify the design and to speed up the writing of traces into the buffer, all information to be stored within each event are a multiple of 32 bits.

Using syscalls like `gettimeofday()` to time-stamp the event introduces too much overhead. We must directly read a register of the machine whenever it is possible. This has been done on IA32 by using the TSC register. This will need to be studied for other architectures (PPC, IA64, ...) and for NUMA[4] machines where each node may have its own counter.

3.3.4 Using the patched NPTL

PTT is made of three parts:

- A patch that adds the PTT trace points into the NPTL routines.
- A patch that adds into NPTL the PTT code that writes the traces into the buffer.
- The code of the daemon and the four PTT commands.

PTT is delivered with instructions explaining how a version of NPTL can be patched and compiled. As explained above, no modification or recompilation of the application is required.

There are two cases for using the patched NPTL:

- For simple programs, it is easy to force the library loader to use the appropriate NPTL library. A script is delivered with PTT.
- For complex programs like JVMs, it is a bit more complex. The `java` command acts as a library loader: it looks at `/proc/self/exe` in order to find its path and name, then it loads libraries (`libjava.so`, ...) based on its path, and finally it reloads itself with `execve()`. So one cannot simply use `ld.so`.

  There are 3 solutions:

  1. If your system glibc is the same version as the patched one, then you can use `LD_PRELOAD`.
  2. You can edit the ELF header in order to change the library loader name/path. Not so easy...
  3. Or you can build a *chroot* environment with the patched library as default glibc.

  If the patched NPTL is delivered with a distribution, then the `LD_PRELOAD` solution seems appropriate.

[4]Non-Uniform Memory Access

3.3.5 Measures and Performances

We have measured the impact of PTT on several applications: GLucas, Volano™Mark[5] and SPECjbb2000[6] for Java, and an unrealistic program performing only calls to the tracing mechanism. We have also compared the impact of PTT with that of the `strace` command. All results are done with the subset of traced NPTL routines that were available in April: Threads, Mutexes, Barriers and CondVars. This means that the following results are preliminary and will probably be different once PTT is finalized.

On average, one call to the PTT trace mechanism leads to 30 bytes of trace data.

• **GLucas** [5] is an HPC[7] program dedicated to proving the primality of Mersenne numbers ($2^q - 1$). It is an open-source C program that implements a specific FFT[8] by means

[5]Volano™ is a trademark of Volano LLC. [6]

[6]SPECjbb® is a registered trademark of the Standard Performance Evaluation Corporation (SPEC®). [7]

[7]High Performance Computing

[8]Fast Fourier Transform

of threads. This is a perfect tool for measuring the impact of PTT: its consumption of multi-threading is much higher than a simple *producer-consumer* model, it can be configured to use as many threads as wanted and it can be launched for a variable amount of time,

• **Volano™Mark** [6] was designed for comparing JVMs when used by the Volano™ chat product. It is a pure Java server benchmark characterized by long-lasting network connections and high thread counts. It is an unofficial Java benchmark that can be configured to use many (thousands) threads for exchanging data between one client and one server by means of sockets. It creates client connections in groups of 20 (a *room*). It is a stress Java program which often makes a JVM crash or hang and which has been used by several studies of Linux performances in the past [9].

• **SPECjbb®2000** [7] is an official SPEC Java benchmark simulating a 3-tier system, mainly the middle tier (business logic and object manipulation). It uses a small number of threads (2 to 3 times the number of processors).

We have made measures on a 2x IA32 machine with 2.8 GHz processors. We observed that the maximum throughput before buffer overflow was obtained with the unrealistic application running one thread: ~1,800,000 traces per second. Due to contention, using more threads led to a lower throughput.

When running **GLucas** with 1000 iterations and with small (2×10^6) to medium (16×10^6) values for the exponent q , we measured that the system and user CPU cost of the daemon was negligible, less than 1 ‰ of the CPU time consumed by GLucas. The throughput of traces ranged between 5,000 and 50,000 traces per second: 40 times lower than the maximum.

When running **Volano™**Mark with 10 rooms, the results depended greatly on the JVM. It appeared that the three main JVMs available on ia32 do not use NPTL in the same way (this may also be due to the fact that only a subset of NPTL routines were traced at that time), leading to quite different results. First, the impact of using the patched NPTL with tracing disabled compared to using the original NPTL is nearly negligible: less than 2% with the fastest JVM, and less than $\sim$ 5% with the slowest one. Second, the impact of running the bench with the patched NPTL with full tracing compared to the original NPTL was about 16% with the fastest JVM and about 47% with the slowest one. Leading to a volume of traces (client + server) that depends on the JVM: from 215 MB to 1,000 MB.

When running **SPECjbb®2000** 65 times with 10 warehouses on a bi-processors machine, the impact of PTT could not be measured since it was lower than the precision of the measure.

We used the **strace** tool for tracing Volano™Mark in two ways. First, when tracing all system calls and only the Volano™ server, the performances were divided by 14.6. Second, when tracing only the calls to the `futex` system call and only the client, the performances were divided by 3.2. Although strace and PTT trace different things, this clearly shows that PTT is much lighter than strace.

3.3.6 Testing

PTT is delivered with a set of tests.

First, there are tests verifying that the features provided by PTT work fine. Examples: a program checks that the `fork()` is correctly handled; another one checks in detail concurrent accesses to the buffer; and a program checks that overloading the buffer and the daemon leads to a nice message warning the end-user that he may loose traces.

Second, there are tests verifying in detail that the traces generated by each patched NPTL routine are correct.

Third, two versions of a *producer-consumer* model have been written, using condvars or semaphores.

GLucas and Java (Volano™Mark) are used for verifying that PTT does not modify the behavior of a large and complex application.

Also the **OPTS** is run in order to check that the PTT-patched NPTL is still compliant with the POSIX Threads standard.

3.4 User Interface

3.4.1 Commands

Several commands are delivered:

`ptt-trace` for launching the application and generating a binary trace file

`ptt-view` for translating the binary trace file into a human or machine readable text format—see Figure 5. (It will enable the end-user to filter the trace. Filters can be applied on: Process Id, Thread Id, name of POSIX Thread Objects, name of Events.)

`ptt-stat` for providing statistics about the use of POSIX Threads objects, etc

`ptt-paje` for translating the binary trace file into a Pajé trace file.

3.4.2 GUI

The analysis of the trace may be very difficult without the help of a graphical tool. Such a tool may simply help the user to navigate through the traces (filter information, find interacting objects, follow the status and the activity of objects, etc.); or it may also display traces in an easier-to-understand graphical way. Both directions are useful, but we decided to focus only on the second one, because we found a sophisticated open-source tool named Pajé that provides nearly all required features without the pain of designing and coding a tool dedicated to PTT.

Pajé [8] was designed for visualizing the traces of a parallel and distributed language (Athapascan) and was developed in a laboratory of the French Research Center IMAG in Grenoble. Pajé is flexible and scalable and can be used quite easily for visualizing the traces of any parallel or distributed system. It can provide views at different scales with different levels of details and one can navigate back and forth in a large file of traces. It is built on the GNUstep [11] platform: an object-oriented framework for desktop application development, based on the OpenStep specification originally created by NeXT—now Apple. Several important companies (France Telecom, ...) have already used Pajé for visualizing complex traces. Pajé is now available in the *sid* (unstable) Debian distribution and soon in the *sarge* (stable) Debian distribution.

We have done preliminary studies and experiments with Pajé, showing that it seems quite easy to produce traces in the format expected by Pajé.

The figure 6 is an example of how a trace could be visualized: the objects (threads, barriers, ...) appear as horizontal bars, with different colors according to their status; and the interactions between objects (when a thread creates or cancels other threads, etc.) are displayed as vertical arrows. The scenario of the example is: the main thread initializes a barrier (count=2) and creates a thread. Then the two threads call

```
Raw machine format:
0.001724:START_USER_FUNC   : 29336 : 0xb7ecb6b0
0.001908:BARRIER_INIT_IN   : 29336 : 0xb7ecb6b0 : 0x8049d28 : (nil) : 2
0.001909:BARRIER_INIT      : 29336 : 0xb7ecb6b0 : 0x8049d28 : 2
0.001909:BARRIER_INIT_OUT  : 29336 : 0xb7ecb6b0 : 0

Text human format:
0.001724 : Pid 29336, Thread 0xb7ecb6b0 starts user function
0.001908 : Pid 29336, Thread 0xb7ecb6b0 enters function pthread_barrier_init.
0.001909 : Pid 29336, Thread 0xb7ecb6b0 initializes barrier 0x8049d28, left=2
0.001909 : Pid 29336, Thread 0xb7ecb6b0 leaves function pthread_barrier_init.
```

Figure 5: An example of a trace written in human or machine readable text formats.

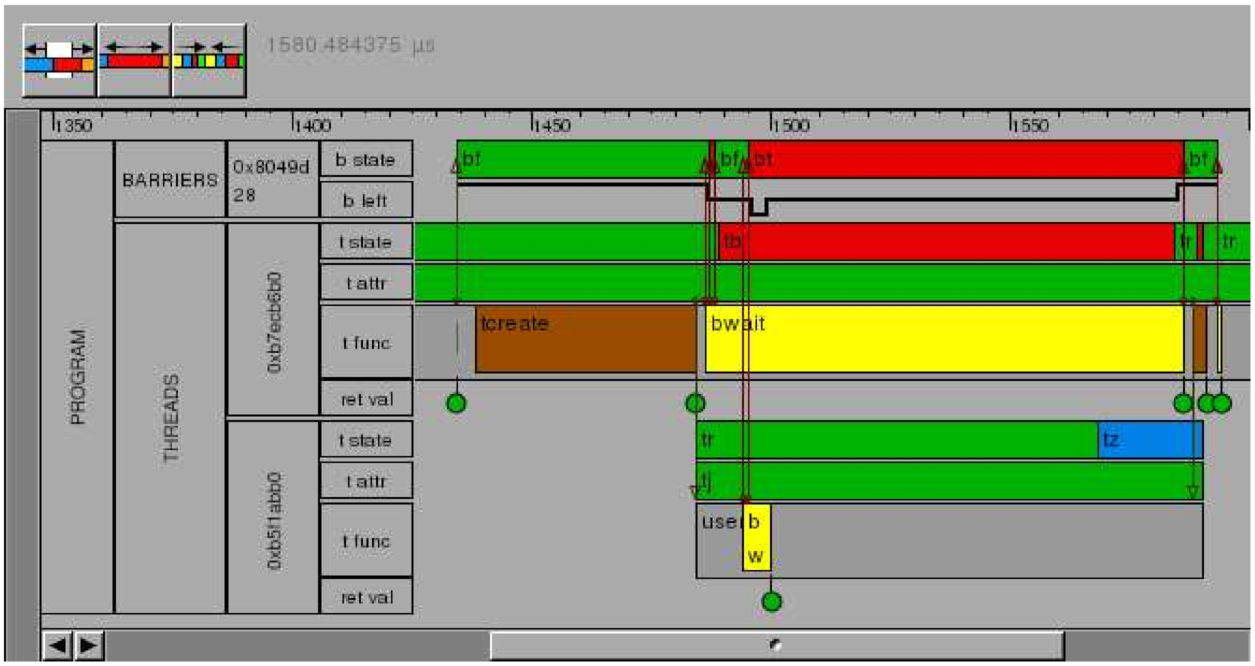

Figure 6: An example of visualizing a PTT trace with Pajé.

`pthread_barrier_wait`: the two threads are freed by the barrier. Finally, the main thread calls `pthread_thread_join` on the second thread and destroys the barrier.

The Pajé tool will enable the user of PTT to clearly see the interactions between the objects involved in his program. Pajé will help the developer of a multi-threaded application to see how his code executes in reality. He will be able to find possible dead-locks, understand which lock is blocking threads thus reducing the performances, and analyze bugs. For analyzing large traces, specific tools (naming, filtering, . . .) must be designed and added in order to manage hundreds of objects and millions of events.

3.5 Status & Future work

Two students work on PTT up to mid July this year. Hereafter, we describe: the status of their work end of April; what they plan to provide in mid July; known limitations; and future potential tasks.

3.5.1 Status in April

At the end of April, PTT already provides the following:

- User and Internal documentations are available.
- PTT is quite reliable and efficient.
- A patch is available for the glibc 2.3.4 (and soon for 2.3.5).

The patch and the sources under CVS are available on SourceForge.net [10].

3.5.2 Expected Status in July

At the end of July, PTT should provide the following:

- Be reliable, efficient, and scalable;
- be available on 3 architectures: IA32, PPC, IA64; handle the most important NPTL objects and routines; provide basic filtering;
- and enable use of Pajé for visualizing small and medium volumes of traces.

3.5.3 Known Limitations

In order to know how much time has elapsed between two events, a time-stamp is recorded within each event. Since this time-stamp is obtained before the event space is reserved in the buffer, it may occur that an event appears in the buffer before older events. Although this could be fixed at the time of decoding the binary trace file, we consider that the error is negligible.

Time-stamping the events on NUMA machines: the actual solution does not take into account the time difference that may appear on such machines.

3.5.4 Next Steps

The main concern when tracing multi-threaded applications is to be able to link the information shown by the trace tool with the traced program. Even with only a dozen threads and mutexes, it is not easy for the user to link the traced thread he is looking at through PTT with the thread managed by his code. Being able to give a name to each instance of NPTL objects is very important. Several ways should be studied and provided in order to replace the internal names (like: `0x401598c0`) by easily understandable names (like: `SocketThread_1`):

- automatically give the thread the name of the routine that was started when the thread was created,
- enable the user to iteratively give names to objects as the user recognizes the objects,
- enable reuse of some existing name table (JVMs).

PTT should be ported on other popular architectures. On machines using several time counters, like NUMA machines, the current version would deliver dates that sometimes could lead to mistakes. This needs to be solved.

Optimizations should be studied: manage the buffer differently; reduce the amount of data stored with each event. More work must be done in order to check the usability and scalability of PTT when used with big and complex applications on large machines with many and fast processors.

We expect people facing complex problems with multi-threaded applications to experiment with PTT, in order to find and fix remaining bugs, and to provide requirements for new features making PTT easier to use and more productive.

PTT could also be a basis for dynamically checking if the application is compliant with the POSIX Thread standard. It is so easy not to fulfill all the constraints of the standard.

The next step of the project is to prove that PTT is really a useful tool: it shortens the time needed for understanding a multi-threaded problem, it speeds up the work of Linux or Java support teams, and it simplifies the analysis of the behavior of NPTL when a misfunction is suspected.

Then PTT could be integrated into Linux Distros. The final goal is to have PTT accepted by the community and then integrated into the GNU libc.

3.5.5 Contributors

PTT has been designed by Sébastien Decugis, Mayeul Marguet, Tony Reix and the developers. The developers are: Nadège Griesser (ENSIMAG-Telecom, Grenoble), Laetitia Kameni-Djinou (UTC, Paris) and Matthieu Castet (ENSIMAG, Grenoble).

4 Conclusion

As we demonstrated in this document, our project has completed some of its objectives, but more work remains pending.

Our testing effort is not complete yet. We have tested only 42 functions of the 150 NPTL contains. Some of the remaining functions may contain bugs or at least are worth testing deeply. The remaining domains are *read-write locks*, *barriers*, *spinlocks*, *thread-specific data*, *timers*, and *message queues*.

Anticipating future problems by writing test cases before someone runs into a bug usually saves a lot of money for everybody. For this reason, we're calling for volunteers to continue our work and complete the testing. This work shall be a continued effort, because the POSIX Standard is changing regularly, therefore if the test suite is not updated regularly it will be deprecated sooner or later. To avoid this situation for the OPTS, the best bet is to have many people use it.

The targeted users are mostly developers of POSIX-compliant implementations. Automating the use of OPTS is easy and, thanks to the TSLogParser tool, collecting and analyzing the results is also quite simple. The next step towards quality for NPTL is to have a real testing process integrated into its development cycle.

The glibc addition to the STP project may be a good solution to solve this, as it is already used for the kernel development and has proved to be useful by detecting new bugs very early in the process.

As we've already told about our Trace Tool, we need more beta testers to try it and give us their comments. This way, we should be able to develop smart tools to use the traces, for example by parsing them in order to find possible problems in threads synchronization or locks contention.

We will also be able to propose our tool to distribution makers, the final goal being that this trace tool be present on all systems. This way, debugging and profiling multi-threaded software will be much easier than it is currently. Is it a utopia? We don't think so…

References

[1] Single UNIX® Specification: `http://www.unix.org/single_unix_specification/`

[2] NPTL Stabilization: `http://nptl.bullopensource.org/`

[3] Austin Revision Group: `http://www.opengroup.org/austin/`

[4] TSLogParser project: `http://tslogparser.sourceforge.net/`

[5] GLucas: `http://www.oxixares.com/glucas/`

[6] Volano™Mark: `http://www.volano.com/`

[7] SPECjbb®2000: `http://www.spec.org/jbb2000/`

[8] Pajé homepage: `http://forge.objectweb.org/projects/paje/`

[9] Linux Kernel Performance Measurement and Evaluation (IBM): `http://linuxperf.sourceforge.net/lwesf-duc_vianney-chat.pdf`

[10] PTT on SourceForge.net: `http://sourceforge.net/projects/nptltracetool/`

[11] GNUstep project: `http://www.gnustep.org/`

[glibc] GNU Lib C: `http://www.gnu.org/software/libc/libc.html`

[OPTS] Open POSIX Test Suite: `http://posixtest.sourceforge.net/`

[LTP] Linux Test Project: `http://ltp.sourceforge.net/`

[STP] Scalable Test Platform: `http://www.osdl.org/lab_activities/kernel_testing/stp`

[PLM] Patch Lifecycle Manager:
`http://www.osdl.org/plm-cgi/plm`

[LTT] Linux Trace Toolkit:
`http://www.opersys.com/LTT/`

[PTT] PTT (NPTL Traces) project:
`http://nptltracetool.sourceforge.net/`