The Native POSIX Thread Library for Linux

Ulrich Drepper
Red Hat, Inc.
drepper@redhat.com

Ingo Molnar
Red Hat, Inc.
mingo@redhat.com

February 21, 2005

This document is completely, utterly out of date when it comes to descriptions of the limitations of the current implementation. Everybody referring to this document to document shortcomings of NPTL is either a moron who hasn't done her/his homework and researched the issue, or is deliberately misleading people as it happens too often with publications done by the Evil from the North-West.

Today's demands for threads can hardly be satisfied by the LinuxThreads library implementing POSIX threads which is currently part of the standard runtime environment of a Linux system. It was not written with the kernel extensions we have now and in the near future available, it does not scale, and it does not take modern processor architectures into account. A completely new design is necessary and this paper will outline the design we came up with.

1 The First Implementation

The LinuxThreads implementation which today is Linux's standard POSIX thread library is based on the principles outlined by the kernel developers at the time the code was written (1996). The basic assumption is that context switches among related proceses are fast enough to handle each user-level thread by one *kernel thread*. Kernel processes can have various degrees of relationship. The POSIX thread specification requires sharing of almost all resources.

Missing thread-aware ABIs for the target architectures, the design did not use thread registers. The thread-local memory was instead located using fixed relationships between the stack pointer and the position of the thread descriptor.

In addition, a *manager thread* had to be created to be able to implement the correct semantics with respect to signals, thread creation, and various other parts of process management.

Perhaps the biggest problem was the absence of usable synchronization primitives in the kernel which forced the implementation to resort to using signals. This, together

with the concept of *kernel thread groups* missing in the kernel versions of the time, lead to non-compliant and fragile signal handling in the thread library.

2 Improvements Over Time

The code of the thread library was significantly improved over the following six years. The improvements come in two areas: the ABI and the kernel.

Newly defined ABI extensions allow the use of thread registers or constructs which can work like registers. This was an essential improvement since now locating the thread-local data is no longer a time consuming operation. Locating thread-local data is essential to almost any operation in the thread library itself. The rest of the runtime environment and the user applications need this as well.

For some architectures the changes were easy. Some had registers set aside for this purpose, others had special processor features which allow storing values in the execution context. But there were some architectures which were left out since they had neither. Those architectures still rely on the method of calculating the thread-local data position based on the stack address. Beside being quite a slow calculation this also means that the APIs which allow the programmer to select the position and size of the stack cannot be implemented for these architectures. These interfaces are especially important when large numbers of threads are used, either at one time or in succession.

The solution used for IA-32 is worth noting since it is not straight-forward and being undoubtedly the most important architecture, this influences the design. The IA-32 architecture, being register starved, has two registers which were not used in the ABI: the segment registers `%fs` and `%gs`. Though not usable for storing arbitrary values they can be used to access memory at arbitrary positions in the virtual memory address space of the process given a fixed offset. The segment register value is used to access some data structures the kernel creates for the processor to access and which contains a base address for each valid segment index. With different base addresses it is possible to access different parts of the virtual address space using the same offset; this is exactly what is needed for thread-local data access.

The problem with this approach is that the segment registers must be supported by some data structures the processor uses. The base address of the segment is stored in a descriptor table. Access to these data structures is limited to operating system itself, not the user level code, which means operations to modify the descriptor table are slow. Context switching between different processes also got slower since additional data structures had to be reloaded at every context switch. Also, the kernel has to handle memory allocation for the table which can be problematic if many entries are needed due to the nature of the memory required (it must be in permanently mapped memory locations which can be difficult to achieve). In addition the number of different values the segment register can have, and therefore the number of different addresses which can be represented, is limited to 8192.

Overall, using “thread registers” brought more speed, flexibility, and API-completeness but it restricted the number of threads and it had negative impacts on the system’s performance.

The changes made in the development up to version 2.4 of the kernel consist of

stabilizing to functionality which allowed to use segment registers on IA-32 as well as improvements to the `clone()` system call which is used to create the kernel threads. These changes could have been used to eliminate some of the requirements for the existence of the manager thread and also provide the correct semantics with respect to the process ID, namely, that the process ID is the same for all threads.

The manager thread could not have been eliminated completely, though, for a number of reasons. One is the fact that the stack memory deallocation could not be performed by the thread which is using the memory itself. A second reason is that to avoid zombie kernel threads the terminated threads must `wait`ed on.

But since these and numerous other problems were not yet solved it was not considered to rewrite the thread library to take advantage of the new features which became available .

3 Problems With The Existing Implementation

The existing implementation has been found to perform reasonably well in many applications but it nevertheless has numerous problems, especially when stressed hard:

- The existence of the manager thread causes problems. If the manager gets killed the remainder of the process is in a state which must be manually cleaned up. Having the manager handle operations like thread creation and cleanup makes it a bottleneck.
- The signal system is severely broken. It does not conform with the behavior POSIX specifies. Sending a signal to the process as a whole is not implemented.
- The use of signals to implement the synchronization primitives cause enormous problems. The latency of the operations is high. And the already complicated signal handling in the thread library gets even more complicated. Spurious wakeups are happening all the time and must be worked around. Beside the problem of misinterpreting a wakeup this also adds even more pressure on the kernel signal system.
- As a noteworthy special case of the broken signal handling the incorrect implementation of `SIGSTOP` and `SIGCONT` should be mentioned. Without the kernel handling these signals correctly the user will for instance not be able to stop a multi-threaded process (e.g., with Control-Z in shells with job handling support). Only one thread would be stopped. Debuggers have the same problem.
- Each thread having a different process ID caused compatibility problems with other POSIX thread implementations. This is relieved in part by the fact that signals couldn't be used very well either but still was noticeable.
- On IA-32 the limit on the number of threads (8192, minus one for the manager) is proving to be a problem for some people. Although threads are often misused in such situations, the offending applications are known to work on other platforms.

On the kernel side there are also problems:

- Processes with hundreds or thousands of threads make the `/proc` file system barely usable. Each thread shows up as a separate process.
- The problems with the signal implementation are mainly due to missing kernel support. Special signals like `SIGSTOP` would have to be handled by the kernel and for all threads.
- The misuse of signals to implement synchronization primitives adds even more to the problems. Delivering signals is a very heavy-handed approach to ensure synchronization.

4 Goals For A New Implementation

Trying to fix the existing implementation is not a worthwhile goal. The whole design is centered around limitations of the Linux kernel at that time. A complete rewrite is necessary. The goal is to be ABI compatible (which is not an unobtainable goal thanks to the way the POSIX thread API is designed). Still it must be possible to reevaluate every design decision made. Making the right decisions means knowing the requirements on the implementation. The requirement which were collected include:

POSIX compliance Compliance with the latest POSIX standard is the highest goal to achieve source code compatibility with other platforms. This does not mean that extensions beyond the POSIX specification are not added.

Effective Use Of SMP One of the main goals of using threads is to provide means to use the capabilities of multi-processor systems. Splitting the work in as many parts as there are CPUs can ideally provide linear speedups.

Low Startup Cost Creating new threads should have very low costs associated to be able to create threads even for small pieces of work.

Low Link-In Cost Programs which are linked with the thread library (directly or indirectly) but don't use threads should not be affected much by this.

Binary compatibility The new library should be binary compatible with the LinuxThreads implementation. Some semantic differences are unavoidable: since the LinuxThreads implementation is not POSIX compliance these areas necessarily change.

Hardware Scalability The thread implementation should run sufficiently well on large numbers of processors. The administrative costs with increasing numbers of processors should not rise much.

Software Scalability Another use of threads is to solve sub-problems of the user application in separate execution contexts. In Java environments threads are used to implement the programming environment due to missing asynchronous operations. The result is the same: enormous amounts of threads can be created. The new implementation should ideally have no fixed limits on the number of threads or any other object.

Machine Architecture Support Designs for big machines have always been a bit more complicated than that for consumer and mainstream machines. Efficient support for these machines requires the kernel and user-level code close to the OS to know details about the machine's architecture. Processors in big machines for instance are often divided in separate nodes and using resources on other nodes is more expensive.

NUMA Support One special class of future machines of interest are based on non-uniform memory architectures (NUMA). Code like the thread library should be designed with this in mind to not defeat the benefits of the architecture when using threads on such machines. The design of data structures is important.

Integration With C++ C++ defines exception handling which deals automatically with the cleanup of objects in the scopes which are left when throwing an exception. Cancellation of a thread is similar to this and it is reasonable to expect that cancellation also calls the necessary object destructors.

5 Design Decisions

Before starting the implementation a number of basic decisions have to be made. They affect the implementation fundamentally.

5.1 1-on-1 vs. M-on-N

The most basic design decision which has to be made is what relationship there should be between the kernel threads and the user-level threads. It need not be mentioned that kernel threads are used; a pure user-level implementation could not take advantage of multi-processor machines which was one of the goals listed previously. One valid possibility is the 1-on-1 model of the old implementation where each user-level thread has an underlying kernel thread. The whole thread library could be a relatively thin layer on top of the kernel functions.

The alternative is a library following the M-on-N model where the number of kernel threads and user-level threads do not have to be in a fixed correlation. Such an implementation schedules the user-level threads on the available kernel threads. Therefore we have two schedulers at work. If they are not collaborating the scheduling decisions made can significantly reduce performance. Various schemes to achieve collaboration have been proposed over the years. The most promissing and most used one is Scheduler Activations. Here the two schedulers work closely together: the user-level scheduler can give the kernel scheduler hints while the kernel scheduler notifies the user-level scheduler about its decisions.

The consensus among the kernel developers was that an M-on-N implementation would not fit into the Linux kernel concept. The necessary infrastructure which would have to be added comes with a cost which is too high. To allow context switching in the user-level scheduler it would be often necessary to copy the contents of the registers from the kernel space.

Additionally many problems the user-level scheduling helps to prevent are no real problems for the Linux kernel. Huge numbers of threads are no issue since the scheduler and all the other core routines have constant execution time ($O(1)$) as opposed to linear time with respect to the number of active processes and threads.

Finally, the costs of maintaining the additional code necessary for an M-on-N implementation cannot be neglected. Especially for highly complicated code like a thread library a lot can be said for a clean and slim implementation.

5.2 Signal Handling

Another reason for using an M-on-N model is to simplify the signal handling in the kernel. While the signal handler, and therefore also the fact, whether a signal is ignored, is a process-wide property, the signal masks are per-thread. With thousands of threads in a process this means that the kernel potentially has to check each and every thread's signal mask to determine whether a signal can be delivered. If the number of kernel threads would be kept low by the M-on-N model the work would be done at the user-level.

Handling the final signal delivery at the user-level has several drawbacks, though. A thread which does not expect a certain signal must not notice that it received a signal. The signal can be noticed if the thread's stack is used for the signal delivery and if the thread receives an `EINTR` error from a system call. The former can be avoided by using an alternate stack to deliver signals. But the `sigaltstack` feature is also available to the user. Allowing both is hard. To prevent unacceptable `EINTR` results from system calls the system call wrappers have to be extended which introduces additional overhead for normal operation.
There are two alternatives for the signal delivery scenario:

1. Signals are delivered to a dedicated thread which does not execute any user code (or at least no code which is not willing to receive all possible signals). The drawbacks are the costs for the additional thread and, more importantly, the serialization of signals. The latter means that, even if the dedicated signal thread distributes the handling of signals to other threads, first all signals are funneled through the signal thread. This is contrary to the intend (but not the words) of the POSIX signal model which allows parallel handling of signals. If reaction time on signals is an issue an application might create a number of threads with the sole purpose of handling signals. This would be defeated by the use of a signal thread.

2. Signals can be delivered to the user level with a different mean. Instead of using the signal handler a separate upcall mechanism is used. This is what would be used in a Scheduler Activation based implementation. The costs are increased complexity in the kernel which would have to implement a second signal delivery mechanism and the userlevel code has to emulate some signal functionality. For instance, if all threads block in a `read` call and a signal is expected to wake one thread up by returning with `EINTR` this must continue to work.

In summary, it is certainly possible to implement the signal handling of a M-on-N implementation at user-level but it is hard, a lot of code, and it is slowing down normal operations.

Alternatively the kernel can implement the POSIX signal handling. In this case the kernel will have to handle the multitude of signal masks but this will be the only problem. Since the signal will only be sent to a thread if it is unblocked no unnecessary interruptions through signals occur. The kernel is also in a much better situation to judge which is the best thread to receive the signal. Obviously this helps only if the 1-on-1 model is used.

5.3 Helper/Manager Thread Or Not

In the old thread library a so-called manager thread was used to handle all kinds of internal work. The manager thread never executes user code. Instead all the other threads send requests like 'create a new thread' which were centrally and sequentially executed by the manager thread. This was necessary to help implementing the correct semantics for a number of problems:

- To be able to react to fatal signals and kill the entire process the creator of a thread constantly has to watch all the children. This is not possible except in a dedicated thread if the kernel does not take over the job.
- Deallocation of the memory used as stacks has to happen after the thread is finished. Therefore the thread cannot do this itself.
- Terminating threads have to be `waited` on to avoid turning them into zombies.
- If the main thread calls `pthread_exit` the process is not terminated; the main thread goes to sleep and it is the job of the manager to wake it once the process terminates.
- In some situations threads need help to handle semaphore operations.
- The deallocation of thread-local data required iterating over all threads which had to be done by the manager.

None of these problem necessarily implies that a manager thread has to be used. With some support in the kernel the manager thread is not necessary at all. With a correct implementation of the POSIX signal handling in the kernel the first item is solved. The second problem can be solved by letting the kernel perform the deallocation (whatever this actually might mean in an implementation). The third item can be solved by the kernel automatically reaping terminated threads. The other items also have solutions, either in the kernel or in the thread library.

Not being forced to serialize important and frequently performed requests like creating a thread can be a significant performance benefit. The manager thread can only run on one of the CPUs, so any synchronization done can cause serious scalability problems on SMP systems, and even worse scalability problems on NUMA systems. Frequent reliance on the manager thread also causes a significantly increased rate of

context-switching. Having no manager thread in any case simplifies the design. The goal for the new implementation therefore should be to avoid a manager thread.

5.4 List of all Threads

The old Linux threads implementation kept a list of all running threads which occasionally was traversed to perform operations on all threads. The most important use is to kill all threads when the process terminates. This can be avoided if the kernel takes care of killing of the threads if the process exist.

The list was also used to implement the `pthread_key_delete` function. If a key is deleted by a call to `pthread_key_delete` and later reused when a following call to `pthread_key_create` returns the same key, the implementation must make sure that the value associated with the key for all threads is `NULL`. The old implementation achieved this by actively clearing the slots of the thread-specific memory data structures at the time the key was deleted.

This is not the only way to implement this. If a thread list (or walking it) has to be avoided it must be possible to determine whether a destructor has to be called or not. One way to implement this is by using generation counters. Each key for thread-local storage and the memory allocated for them in the thread's data structures, have such a counter. When a key gets allocated the key's generation counter is incremented and the new value is assigned to the counter in the thread data structure for the key. Deleting a key causes the generation counter of the key to be incremented again. If a thread exits only destructors for which the generation counter of the key matches the counter in the thread's data structure are executed. The deletion of a key becomes a simple increment operation.

Maintaining the list of threads cannot be entirely avoided. To implement the `fork` function without memory leaks it is necessary that the memory used for stacks and other internal information of all threads except the thread calling `fork` is reclaimed. The kernel cannot help in this situation.

5.5 Synchronization Primitives

The implementation of the synchronization primitives such as mutexes, read-write locks, conditional variables, semaphores, and barriers requires some form of kernel support. Busy waiting is not an option since threads can have different priorities (beside wasting CPU cycles). The same argument rules out the exclusive use of `sched_yield`. Signals were the only viable solution for the old implementation. Threads would block in the kernel until woken by a signal. This method has severe drawbacks in terms of speed and reliability caused by spurious wakeups and derogation of the quality of the signal handling in the application.

Fortunately some new functionality was added to the kernel to implement all kinds of synchronization primitives: futexes [Futex]. The underlying principle is simple but powerful enough to be adaptable to all kinds of uses. Callers can block in the kernel and be woken either explicitly, as a result of an interrupt, or after a timeout.

A mutex can be implemented in half a dozen instruction with the fast path being entirely at user-level. The wait queue is maintained by the kernel. There are no fur-

ther user-level data structures needed which have to be maintained and cleaned up in case of cancellation. The other three synchronization primitives can equally well be implemented using futexes.

Another big benefit of the futex approach is that it works on shared memory regions and therefore futexes can be shared by processes having access to the same piece of shared memory. This, together with the wait queues being entirely handled by the kernel, is exactly the requirement the inter-process POSIX synchronization primitives have. Therefore it is becomes now possible to implement the often asked for `PTHREAD_PROCESS_SHARED` option.

5.6 Memory Allocation

One of the goals for the library is to have low startup costs for threads. The biggest problem time-wise outside the kernel is the memory needed for the thread data structures, thread-local storage, and the stack. Optimizing the memory allocation is done in two steps:

- The necessary memory blocks are merged. I.e., the thread data structures and the thread-local storage are placed on the stack. The usable stack array starts just below (or above in case of an upward growing stack) the memory needed for the two.

  In the thread-local storage ABI defined in the ELF gABI requires only one additional data structure, the DTV (Dynamic Thread Vector). The memory needed for it might vary and therefore cannot be allocated statically at thread start time.

- The memory handling, especially the de-allocation, is slow. Therefore major reductions can be achieved if the whole allocation can be avoided. If memory blocks are not freed directly when the thread terminates but instead is kept around this is exactly what happens. An `munmap` of the stack frame causes expensive TLB operations, e.g., on IA-32 it causes a global TLB flush, which can also be broadcasted to other CPUs. Hence the caching of stack frames is a key step toward good thread-create and thread-exit performance.

  Another advantage is that at the time a thread terminates some of the information in the thread descriptor is in a useful state and does not have to be re-initialized when the descriptor gets reused.

  Especially on 32-bit machines with their restricted address space it is not possible to keep unlimited memory around for reuse. A maximum size for the memory cache is needed. This is a tuning variable which on 64-bit machines might as well have a value large enough to never be exceeded.

This scheme works fine most of the time since the threads in one process often have only a very limited number of different stack sizes.

The only drawback of this scheme is that since the thread handle is simply the pointer to the thread descriptor successively created threads will get the same handle. This might hide bugs and lead to strange results. If this really becomes a problem the thread descriptor allocation can get a debug mode in which it avoids producing the

same thread handles again. This is nothing the standard runtime environment should be troubled with.

6 Kernel Improvements

Even the early 2.5.x development version of the Linux kernel did not provide all the functionality needed for a good thread implementation. The changes added to the official kernel version since that then include the following. All these changes were made in August and September 2002 by Ingo Molnar as part of this project. The design of the kernel functionality and thread library went hand in hand the whole time to ensure optimal interfaces between the two components.

- Support for an arbitrary number of thread-specific data areas on IA-32 and x86-64 through the introduction of the TLS system call. This system call allows to allocate one or more GDT (Global Descriptor Table, a CPU data structure) entries which can be used to access memory with a selected offset. This is an effective replacement for a thread register. The GDT data structure is per-CPU and the GDT entries per-thread, kept current by the scheduler.

  This patch enabled the implementation of the 1-on-1 threading model without limitation on the number of threads since with the previously used method (via the LDT, local descriptor table, CPU data structure) the number of threads per process were limited to 8192. To achieve maximal scalability without this new system call an M-on-N implementation would have been necessary.

- The `clone` system call was extended to optimize the creation of new threads and to facilitate the termination of threads without the help of another thread (the manager thread fulfilled this role in the previous implementation). In the new implementation the kernel stores in a given memory location the thread ID of the new thread if the `CLONE_PARENT_SETTID` flag is set and/or clears the same memory location once the thread is terminated if the `CLONE_CLEARTID` flag is set. This can be used by user-level memory management functionality to recognize an unused memory block. This helps implementation user-level memory management without requiring the kernel to know any details about it.

  Furthermore the kernel does a futex wakeup on the thread ID. This feature is used by the `pthread_join` implementation.

  Another important change is adding support for signal safe loading of the thread register. Since signals can arrive at any time Either they have to be disabled around the `clone` call or the new thread must be started by the kernel with the thread register already loaded. The latter is what another extension to `clone` implements, the `CLONE_TLS` flag. The exact form of the parameter passed to the kernel is architecture specific.

- The POSIX signal handling for multi-threaded processes is now implemented in the kernel. Signals sent to the process are now delivered to one of the available

thread of the process. Fatal signals terminate the entire process. Stop and continue signals affect the entire process; this enables job control for multi-threaded process, something dearly missed in the old implementation. Shared pending signals are supported as well.

- A second variant of the `exit` system call was introduced: `exit_group`. The old system call kept the meaning of terminating the current thread. The new system call terminates the entire process.

  At the same time the implementation of the `exit` handling was significantly improved. The time to stop a process with many threads takes now only a fraction of what it used to. In one instance starting and stopping 100,000 threads formerly took 15 minutes; this is now takes 2 seconds.

- The `exec` system call now provides the newly created process with the process ID of the original process. All other thread in the process are terminated before the new process image gets control.

- Resource usage reported to the parent (CPU and wall time, page faults etc) are reported for the entire process and not just the initial thread.

- The creation of the content of the `/proc` directory was changed to not include any thread except for the initial one. The initial thread represents the process. This is a necessary measure since otherwise the thousands or tens of thousands of threads a process can have would make the user of `/proc` slow at best.

- Support for detached threads, for which no `wait` has to be performed by the joining thread, joining is done via the futex wakeup done by the kernel upon thread exit.

- The kernel keeps the initial thread around until every thread has exited. This ensures the visibility of the process in `/proc`, and ensures signal delivery as well.

- The kernel has been extended to handle arbitrary number of threads. The PID space has been extended to a maximum of 2 billion threads on IA-32, and the scalability of massively-threaded workloads has been improved significantly. The `/proc` filesystem had to be fixed to support more than 64k processes.

- The way the kernel signals termination of a thread makes it possible for `pthread_join` to return after the child is really dead. I.e., all TSD destructors ran and the stack memory can be reused which is important if the stack was allocated by the user.

7 Results

This section presents the results of two completely different measurements. The first set is a measurement of the time needed for thread creation and destruction. The second measurement concerns itself with measuring the handling of lock contention.

Thread Creation and Destruction Timing

What is measured is simply the time to create and destroy threads, under various conditions. Only a certainly, variable number of threads exist at one time. If the maximum number of parallel threads is reached the program waits for a thread to terminate before creating a new one. This keeps resource requirements at a manageable level. New threads are created by possibly more than one thread; the exact number is the second variable in the test series.

The tests performed were:

for 1 to 20 toplevel threads creating new threads

create for each toplevel thread up to 1 to 10 children

The number of times we repeated the thread creation operation is 100,000 – this was only done to get a measurable test time and should not be confused with earlier 'start up 100,000 parallel threads at once' tests.

The result is table with 200 times. Each time is indexed with the number of toplevel threads and the maximum number of threads each toplevel thread can create before having to wait for one to finish. The created threads do no work at all, they just finish.

We summarize the result of the benchmark runs in two tables. In both cases we flatten one dimension of the measurement result matrix with a minimal function.

Figure 1 shows the result for the different number of toplevel threads creating the actual threads we count. The value used is the minimal time required of all the runs with different numbers of threads which can run in parallel.

What we can see is the NGPT is indeed a significant improvement over LinuxThreads; NGPT is twice as fast. The thread creation process of LinuxThreads was really complicated and slow. What might be surprising is that a difference to NPTL is so large (a factor of four).

The second summary looks similar. Figure 2 shows the minimum time needed based on the number of toplevel threads. The optimal number of threads which are used by each toplevel thread determines the time.

In this graph we see the scalability effects. If too many threads in parallel try to create even more threads all implementations are impacted, some more, some less.

Contention Handling

Figure 3 shows timings of a program which creates 32 threads and a variable number of critical regions which the threads try to enter, a total of 50,000 times [csfast]. The fewer critical regions exist the higher the probability of contention.

The graph shows significant variantions even though the numbers are averages over 6 runs. These differences are caused by scheduling effects which affect all programs which do not do any real work and instead spend all times creating scheduling situations (like blocking on a mutex).

The results for the two kernel versions show that

- the times for NPTL are significantly lower than those for LinuxThreads;

- the 2.4.20-2.21 kernel has a scheduler which was changed to handle the new situations the frequent use of futexes create. Similar changes will be made for the 2.5 development kernel. The message from this development is that tuning of the kernel scheduler and necessary and provides significant gains. There is no reason to believe the code in 2.4.20-2.21 is in any way optimal;
- the expected asymptotic behavior is visible.

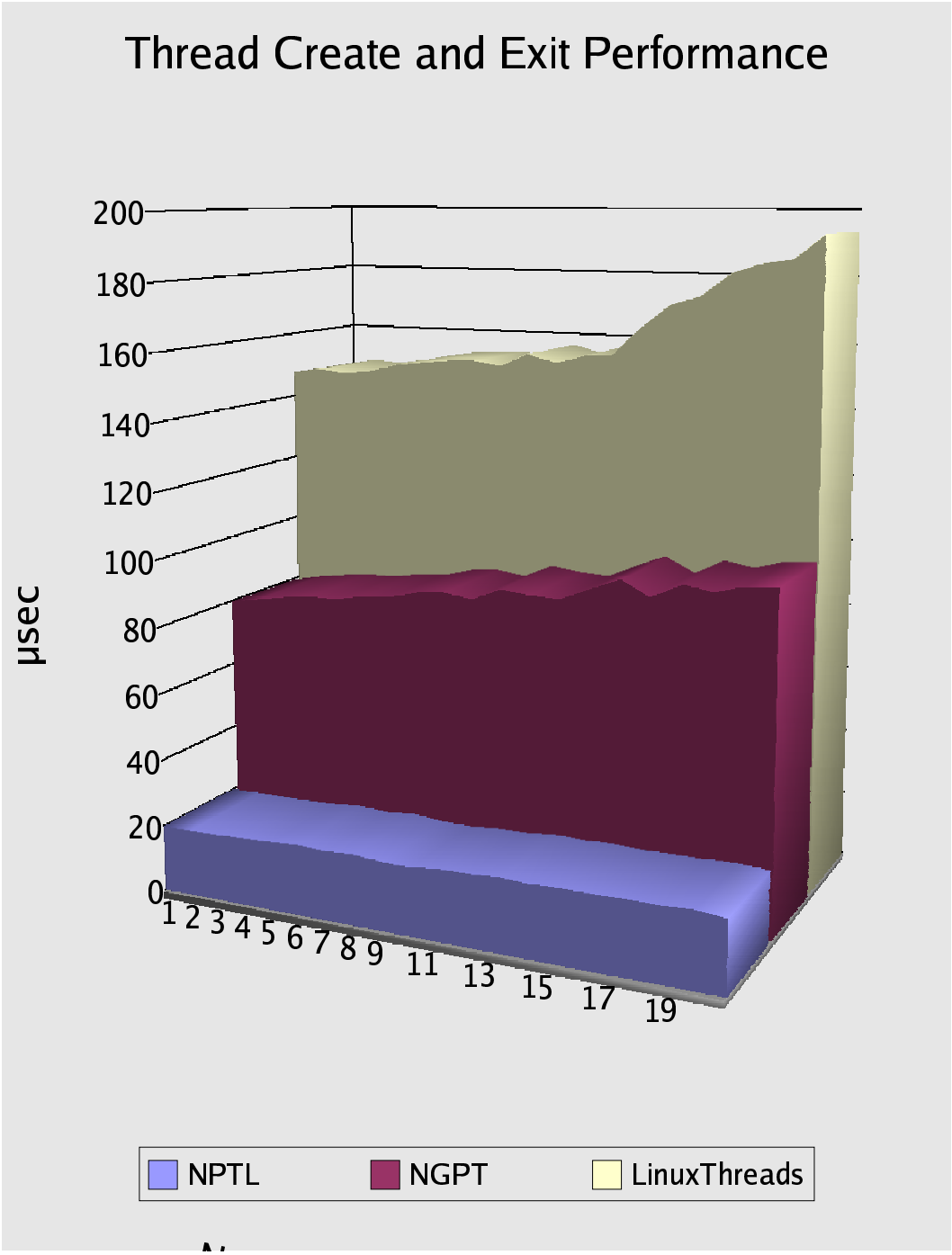

Figure 1: Varying number of Toplevel Threads

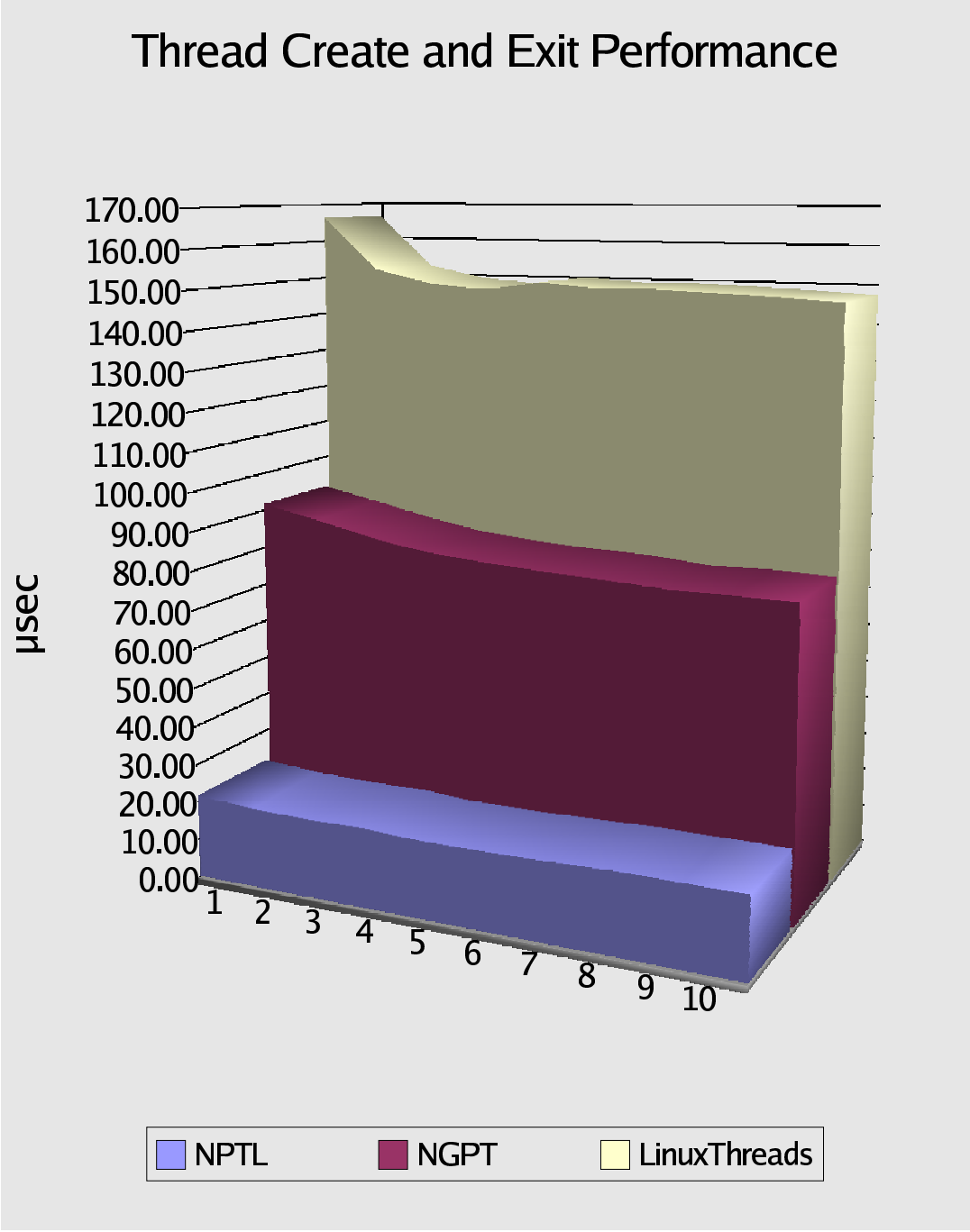

Figure 2: Varying number of Concurrent Children

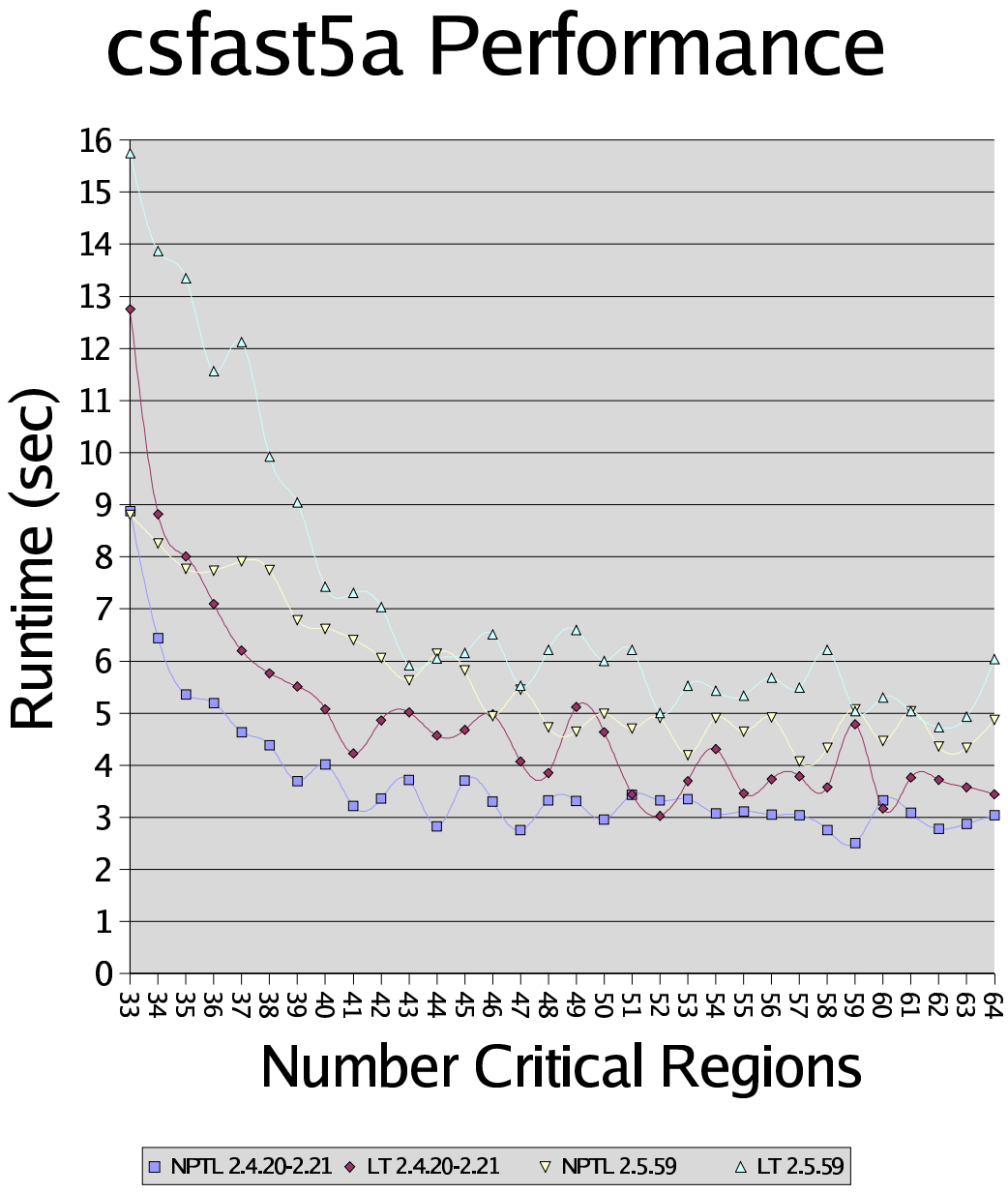

Figure 3: Lock Contention Handling

8 Remaining Challenges

A few challenges remain before 100% POSIX compliance can be achieved. Which of these derivations will depend on how well a solution fits into the Linux kernel implementation.

The `setuid` and `setgid` families of system calls must affect the entire process and not just the initial thread.

The `nice` level is a process-wide property. After adjusting it all threads in the process must be affected.

The CPU usage limit which can be selected with `setrlimit` limits the time spent by all threads in the process together.

Realtime support is mostly missing from the library implementation. The system calls to select scheduling parameters are available but they have no effects. The reason for this is that large parts of the kernel do not follow the rules for realtime scheduling. Waking one of the threads waiting for a futex is not done by looking at the priorities of the waiters.

There are additional places where the kernel misses appropriate realtime support. For this reason the current implementation is not slowed down by support for something which cannot be achieved.

The library implementation contains a number of places with tunable variables. In real world situations reasonable default values must be determined.

A References

This paper is available at http://people.redhat.com/drepper/nptl-design.pdf.

[Futex] Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux, *Hubertus Franke, Rusty Russell, Matthew Kirkwood, Proceedings of the Ottawa Linux Symposium, 2002.*

[TLS] ELF Handler For Thread-Local Storage, *Ulrich Drepper, Red Hat, Inc.*, http://people.redhat.com/drepper/tls.pdf.

[csfast] csfast5a at http://www-106.ibm.com/developerworks/linux/library/l-rt10/ index.html?t=gr,lnxw01=ConSwiP2