The Native POSIX Thread Library for Linux

February 2003

by: Ulrich Drepper and Ingo Molnar

Abstract

The Linux Threads library which is currently part of the standard runtime environment of a Linux system does a poor job of satisfying the demand for POSIX compliant threading used in modern systems. It is not designed to leverage the Linux kernel extensions present and in development today, it does not scale efficiently, and it does not take modern processor architectures into account. A completely new design is necessary and this paper will outline the design and implementation.

redhat

Table of Contents

The Initial Implementation

The Linux Threads implementation, which is currently the standard POSIX thread library for Linux, is based on the principles outlined by the kernel developers at the time the code was written in 1996. The basic assumption was that context switches among related processes would be fast enough to handle each user-level thread by one kernel thread. Kernel processes can have various degrees of relationships for sharing. The POSIX thread specification requires sharing of almost all resources between threads.

Missing thread-aware Application Binary Interface Specifications (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.

A manager thread was required to implement correct semantics for 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. The absence of a concept for thread groups in the kernel versions of the time led to non-compliant and fragile signal handling in the thread library.

Improvements Over Time

The code of the thread library was significantly improved in the six years following its creation. The improvements came in two principal areas: the ABI and the kernel.

Newly defined ABI extensions allowed the use of thread registers or constructs, which can work like registers. This was an essential improvement making locating thread-local data a less time consuming operation. Locating thread-local data is essential to almost any operation in the thread library itself. This data is used by the rest of the runtime environment and user applications as well.

For some CPU architectures the changes were easy. Some had registers specifically set aside for this purpose, while others had special processor features which allowed storing values in the execution context. But some architectures were left out since they had neither. Those architectures still defered to the method of calculating the thread-local data position based on the stack address. Besides being a time costing calculation, the APIs that allow the programmer to select the position and size of the stack cannot function correctly 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 Intel's 32-bit CPU Architecture (IA-32) is worth noting since it is not straight-forward and, as undoubtedly the most prevalent microcomputer architecture, it influences the design. IA-32, being register starved, has two registers which were not used in the original ABI: the segment registers %fs and %gs. Though not generally 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 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 the operating system itself and not the user level code, which means operations to modify the descriptor table are slow. Context switching between different processes is also slower since additional data structures must be reloaded at every context switch. Additionally, the kernel must handle memory allocation for the table which can be problematic if many entries are needed due to the limited nature of the memory required (it must be in permanently mapped memory). Finally, 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 Applications Programming Interface (API) completeness but restricted the number of threads and had negative impacts on the system's overall performance.

The changes made in the development of the kernel up to version 2.4 consisted of stabilizing the functionality which allowed use of the IA32 segment registers as well as improvements to the `clone` system call which is used to create the kernel threads. These changes eliminate some of the requirements for the existence of the manager thread and also provide the correct process ID semantics as the process ID is the same for all threads.

Unfortunately, the manager thread still could not be eliminated completely for a number of reasons. One reason is that the stack memory deallocation could not be performed by the thread which uses the memory itself. A second reason is that terminated threads must be waited on in order to avoid zombie kernel threads. Since these and numerous other problems were not yet solved, there was limited incentive to rewrite the thread library to take advantage of the new features which became available.

Problems with the Existing Implementation

The existing Linux Threads implementation has been found to perform reasonably well in many applications;nevertheless, it has numerous problems, especially when operated in extreme circumstances:

- The existence of the manager thread causes problems. If the manager thread 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 could not be implemented.

- The use of signals to implement the synchronization primitives causes 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 frequent and must be distinguished from normal wakeup conditions and handled appropriately. In addition to the probability of misinterpreting a wakeup, this adds additional pressure on the kernel signal system.

- The incorrect implementation- of SIGSTOP and SIGCONT is a noteworthy case of broken signal handling. Without the kernel handling these signals correctly the user cannot stop a multi-threaded process (e.g., with Control-Z in shells with job handling support). In that case only one thread is stopped. Debuggers make use of this interface and therefore have similar problems.

- Each thread having a different process ID causes compatibility problems with other POSIX thread implementations. This is in part a moot point since signals can'tbe used very well but is still noticeable.

- On IA-32, the artificial limit on the number of threads (8192, minus one for the manager) has proven troublesome. 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 render 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.

Goals For A New Implementation

Trying to incrementally fix the existing implementation would not have been efficient. The whole design is centered around limitations of 1996-era Linux kernel technology. A complete rewrite taking into account modern kernel features was necessary. The goal was to be ABI compatible with Linux Threads,which is not an unobtainable goal thanks to the way the old thread API was designed. Still it was necessary to reevaluate every design decision made. Making the right decisions meant knowing the requirements of the implementation. The requirements 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, but rather that POSIX compliance must take precedence.

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 associated costs so that it's possible to create threads even for small pieces of work.

Low link-in cost

Programs linked with the thread library (directly or in-directly) but not using threads should be minimally affected.

Binary compatibility

The new library should be maximally binary compatible with the Linux Threads implementation. Some semantic differences are unavoidable as the Linux Threads implementation is not POSIX compliant; the non-compliant functionality necessarily must change.

Hardware Scalability

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

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 mainframe machines have always been more complicated than those 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. For instance, processors in these machines are often divided into 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 leverage the benefits of NUMA when using threads on such machines. For these systems the design of data structures is critical.

Integration With C++

C++ defines exception handling, which deals automatically with the cleanup of objects in the scopes 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.

Design Decisions

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

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

The most basic design decision involves the relationship 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 exist, but it would not be able to take advantage of multi-processing, which was one of the primary goals listed previously.

One valid possibility is the 1-on-1 model of the Linux Threads 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 most popular 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. In this case we actually have two schedulers at work. Since there is no default mechanism for explicit collaboration between the two schedulers, the independent scheduling decisions can significantly reduce performance. Various schemes to achieve collaboration have been proposed. The most promising and most used 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 coordinated scheduling infrastructure that must 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 of the problems the user-level scheduling helps to prevent are not real problems for the Linux kernel. Huge numbers of threads are not a significant 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, there's a lot to be said for a clean and slim implementation.

Signal Handling

Another reason for using an M-on-N model is to simplify the signal handling in the kernel. Signal masks are maintained on a per-thread basis whereas the registration of a signal handler, and therefore also the fact whether a signal is ignored, is a process wide property. With large numbers of threads in a process the kernel potentially must check 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 equivalent work would be done at the user level.

Handling the final signal delivery at the user-level has several drawbacks. A thread which does not expect a certain signal must not be able to detect that it received a signal. The signal would be noticeable if the thread's stack were used for the signal delivery or if the thread received an interrupted by signal (EINTR) error from a system call. The former situation can be avoided by using an alternate stack to deliver signals but adds complexity since the use of an alternate stack is also available to the user through the `sigaltstack` call. 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 include 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 all signals are funneled through the dedicated signal thread. This is contrary to the intent (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 could also be delivered to the user level by a different means. Instead of the signal handler, a separate up call 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 required emulation of some signal functionality by the user level code. For instance, if all threads block in a read call and a signal is expected to wake one thread by returning with `EINTR`, this thread must receive the return code and continue processing.

In summary, it is certainly possible to implement the signal handling of a M-on-N implementation at user-level, but it adds complexity, bulk, and unnecessary delay.

Alternatively all POSIX signal handling can be implemented in the kernel. In this case the kernel must resolve the multitude of signal masks, but the implementation is otherwise straightforward. 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 determine the best thread to receive the signal. Obviously this helps only if the1-on-1 model is used.

Helper/Manager Thread or Not

In the current Linux Threads library a so-called manager thread is used to handle a variety of internal work. The manager thread never executes user program code. Instead all the other threads send requests like 'create a new thread' which are centrally and sequentially executed by the manager thread. This is necessary to help implement 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 deallocate it's own stack.
- 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 requires iterating over all threads which has to be done by the manager.

None of these problems necessarily implies that a manager thread must 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's 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.

List of all Threads

The Linux Threads implementation keeps a list of all running threads which is occasionally traversed to perform operations involving all threads. The most important operation is killing all threads at process termination. This could be avoided if the kernel were responsible for killing the threads when the process exits.

The thread list is 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 Linux Threads implementation achieves this by walking the list to actively clear the slots of the thread-specific memory data structures at the time the key is deleted.

This is a cumbersome implementation of `pthread_key_delete`. If a thread list (or walking it) has to be avoided, it must be possible to determine whether a destructor must be called. One possible implementation involves generation counters. Each key for thread-local storage and the associated memory in the thread's data structures would have such a counter. Upon allocation the key's generation counter would be incremented and the new value assigned to the counter in the thread data structure for the key. Deleting a key also causes the key's generation counter to be incremented. On exit it is only necessary to execute the 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 therefore becomes a simple increment operation.

Maintaining the list of threads can not be entirely avoided. Implementation of the `fork` function without memory leaks requires reclaiming the memory used for stacks and other internal information of all threads except the thread calling `fork`. The kernel can not help in this situation.

Synchronization Primitives

The implementation of the synchronization primitives such as mutexes, read-write locks, condition variables, semaphores, and barriers requires some form of kernel support. Busy waiting is inefficient and fails to account for differences in thread priorities. The same arguments rule out the exclusive use of sched yield. Signals were the only viable solution for the Linux Threads implementation. Threads 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 degradation 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 in the form of futexes [Futex]. The underlying principle is simple but powerful enough to adapt 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.

For example, a mutex can be implemented in half a dozen instructions with the fast path being entirely at user-level. The wait queue is maintained by the kernel. There are no further user-level data structures needed which have to be maintained and cleaned up in case of cancellation. The other three synchronization primitives can be equally well implemented using futexes.

Another benefit of the futex approach is that it works on shared memory regions and can therefore 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. It now becomes possible to implement the desired `PTHREAD_PROCESS_SHARED` option.

Memory Allocation

One of the goals for the library is to have minimal startup costs for threads. The biggest time consuming operation outside the kernel is allocating the memory needed for the thread data structures, thread-local storage, and the stack. Optimizing this 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.

- Since the memory handling, especially the de-allocation, is slow, major improvements can be achieved by avoiding unnecessary allocation and deallocation. An `munmap` of the stack frame causes expensive translation look-aside buffer (TLB) operations, e.g., on IA-32 it causes a global TLB flush, which must also be broadcast to other CPUs on SMP systems. If memory blocks are kept for reuse and not freed directly when the thread terminates, the number of allocation/deallocation cycles is reduced. Hence the caching of stack frames is a key step toward good thread-create and thread-exit performance.

  A by-product 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.

  It is not possible to keep unlimited memory around for reuse, especially on 32-bit machines due to their restricted address space. 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.

One potential drawback of this scheme is that the thread handle is simply the pointer to the thread descriptor, so successively created threads will get the same handle. This might hide bugs and lead to strange results. If this became a significant problem, the thread descriptor allocation routine could have a debug mode in which it would avoid producing the same thread handles again. This is nothing the standard runtime environment should be troubled with.

Kernel Improvements

The early 2.5.x development version of the Linux kernel provided only a portion of the functionality needed for a good thread implementation. Additional changes to the official kernel 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 to ensure optimal interfaces between the two components. Changes to the kernel include:

- 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 allocates 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 are kept current by the scheduler.

  The TLS patch enabled the implementation of the 1-on-1 threading model without limitation on the number of threads. The previously used method (via the LDT, local descriptor table, CPU data structure) had limited the number of threads per process 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 Linux Threads implementation. In the new implementation the kernel stores the thread ID of the new thread in a given memory location if the `CLONE_PARENT_SETTID` flag is set. This implementation also clears this 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 enable implementation of user-level memory management without kernel intervention.

  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 a signal safe loading of the thread register. Since signals can arrive at any time they either have to be disabled for the duration of the `clone` call, or the new thread must be started with the thread register already loaded. The latter is what another extension to `clone` implements, via 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 threads of the process. Fatal signals terminate the entire process. Stop and continue signals affect the entire process, enabling job control for multi-threaded processes, a desirable feature missing in the Linux Threads implementation. Shared pending signals are also supported.

- 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.

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

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

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

- The `/proc` directory implementation was improved to cope with the potentially thousands of entries resulting from the threads in all the processes. Each thread has its own subdirectory but all names, except those of the main thread, start with a dot and are therefore not visible in the normal output of `ls`. The overall implementation of the `/proc` filesystem has been improved and optimized to handle huge amounts of data

- Support for detached threads, for which no wait has to be performed by the joining thread. This join is now implemented via a futex wakeup in the kernel upon thread exit.

- The kernel maintains the initial thread 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 an 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` file system can now 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 therefore stack memory can be reused, which is important if the stack was allocated by the user.

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. The first variable in this test is the number of threads which 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 top level threads creating new threads

create for each top level 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 tests that tended to start up 100,000 parallel threads at once.The result is a table measuring 200 execution times. Each time is indexed with the number of top level threads and the maximum number of threads each top level thread can create before having to wait for one to finish. The created threads do no work at all, they just finish.

The results of the benchmark runs are summarized in two tables. In both cases we flatten one dimension of the measurement result matrix with a minimal function. Figure 1 (see page 18) shows the result for the different number of top level 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 that NGPT is indeed a significant improvement over Linux Threads; NGPT is twice as fast. The thread creation process of Linux Threads was really complicated and slow, so it is surprising that a difference to NPTL is so large (a factor of four).The second summary looks similar. Figure 2 (see page19) shows the minimum time needed based on the number of top level threads. The optimal number of threads which are used by each top level thread determines the time.

In Figure 2 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 (see page 20) 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 that exist, the higher the probability of contention.

Figure 3 shows significant variations even though the numbers are averages over 6 runs. These differences are caused by scheduling effects which affect all programs. These threads are not responsible for real work and instead spend all of their execution time 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 Linux Threads.
- the 2.4.20-2.21 kernel has a scheduler which was changed to handle new situations that 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 is 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.

Remaining Challenges

A few challenges remain before 100% POSIX compliance can be achieved. The selection of a remedy path 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. Real time support is mostly missing from the library implementation. The system calls to select scheduling parameters are available but they have no guaranteed effect as large parts of the kernel do not follow the rules for real time scheduling. For instance, waking one of the threads that is waiting for a futex is done without looking at the priorities of the threads in the queue. There are additional places where the kernel misses appropriate real time support. For this reason the NPTL does not attempt to support something which cannot be achieved at the kernel level.

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

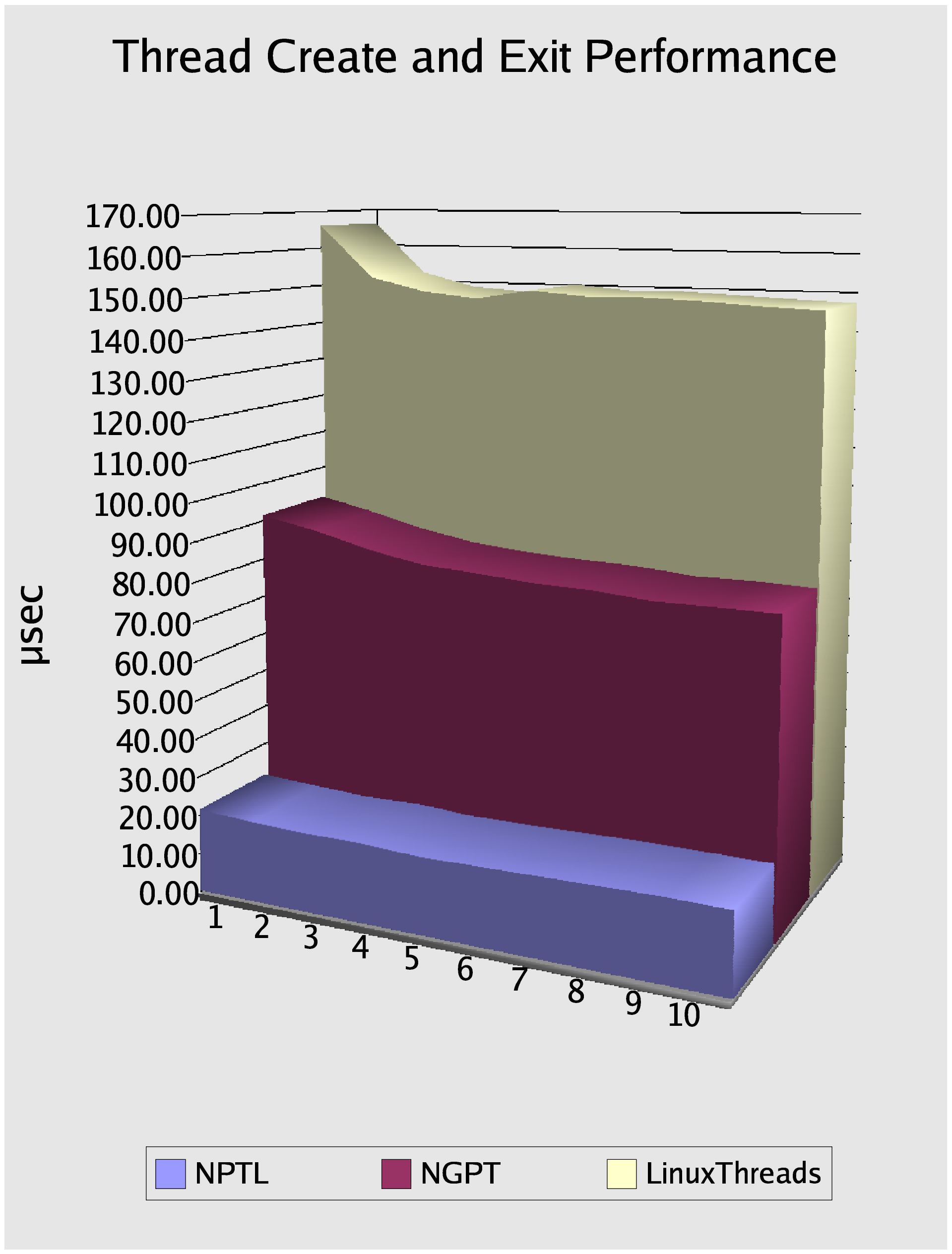

Figure 1: Varying number of Top level Threads

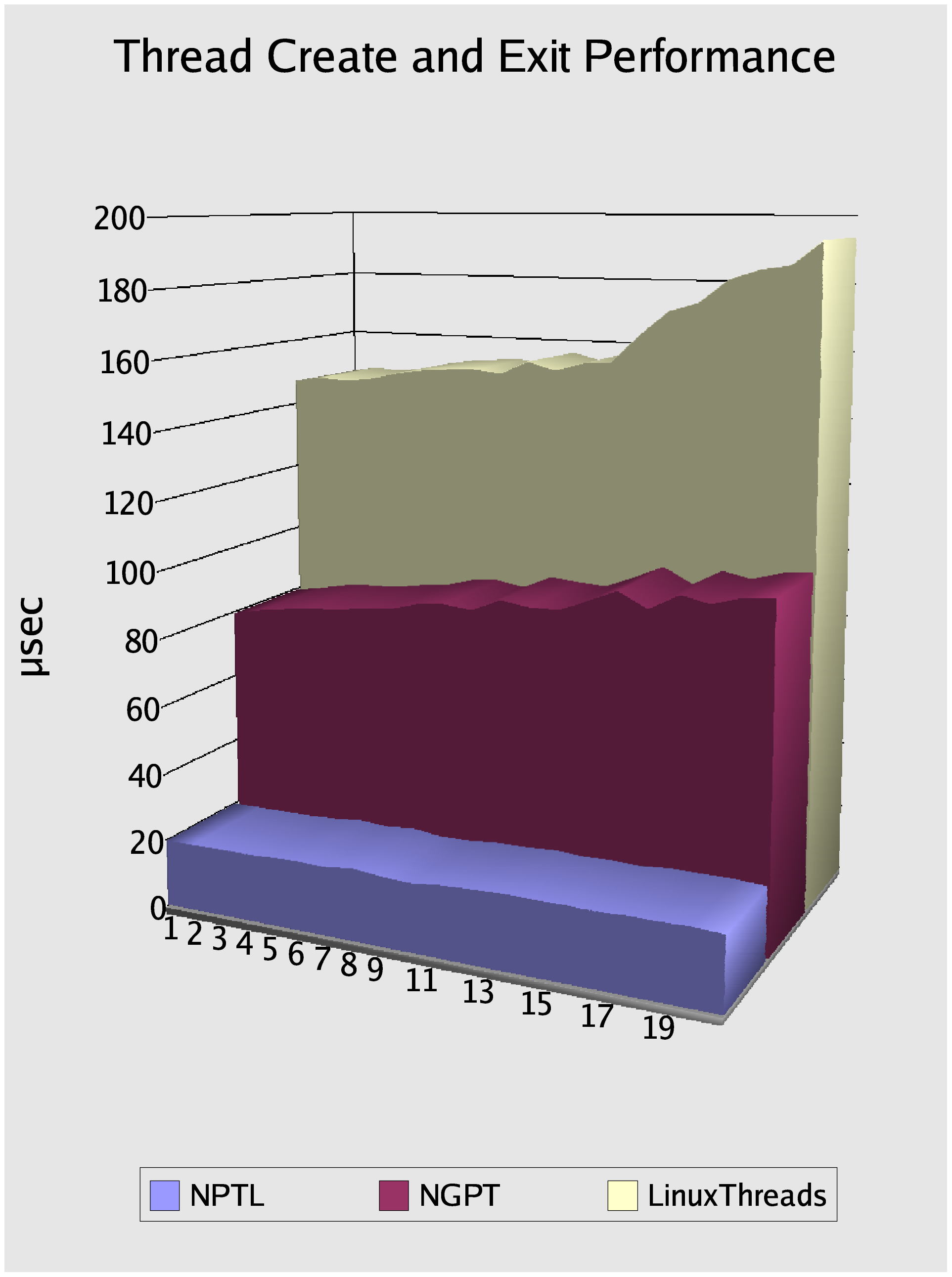

Figure 2: Varying number of Concurrent Children

csfast5a Performance

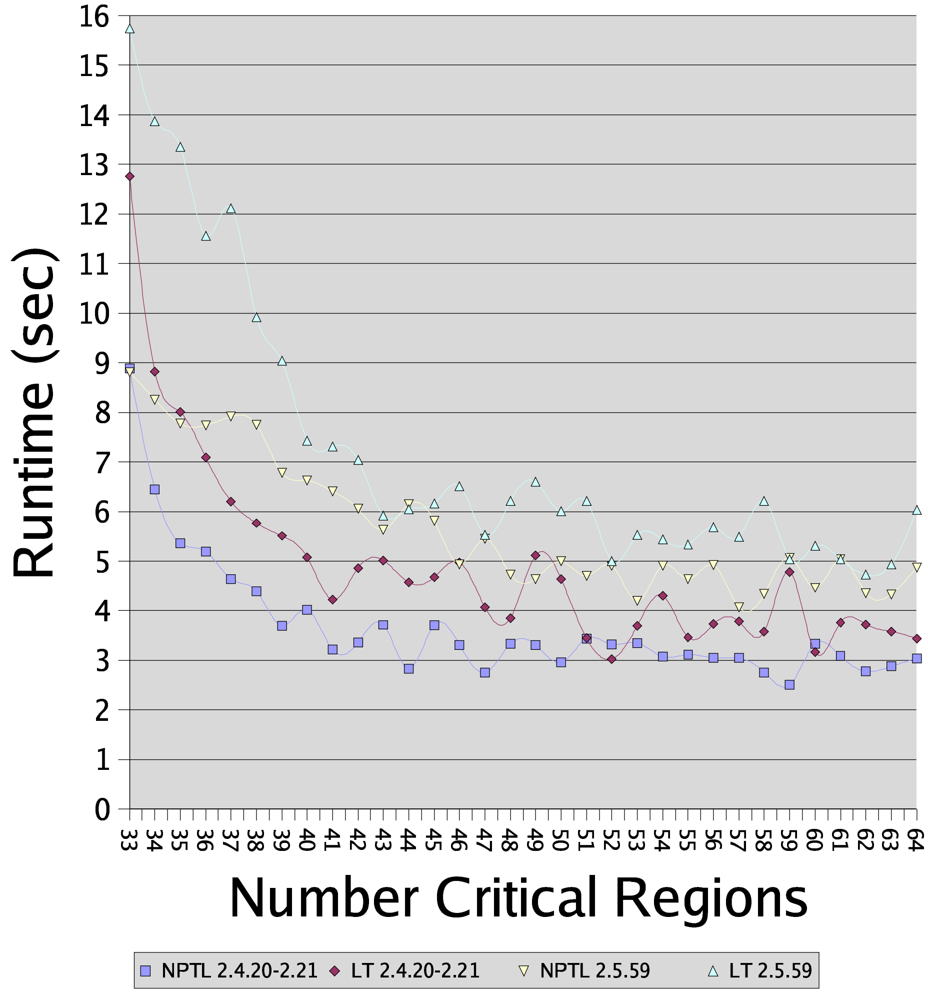

Figure 3: Lock Contention Handling