rthreads: A New Thread Implementation for OpenBSD

Ted Unangst

Abstract

The next generation thread library for OpenBSD will be rthreads. Based on the rfork() system call, rthreads improve the performance, robustness, and scalability of OpenBSD's thread support. In contrast to other recent threading models introduced to BSD systems, rthreads is not based on scheduler activations.

The existing userland pthreads has carried us a long way but it's been showing its age recently. As more applications place more demanding requirements on the thread library its shortcomings become more apparent. This paper will explain these problems, highlight how rthreads resolve them, and then continue with an overview of the rthreads implementation.

Threads

Briefly, threading opens up a new programming model for a developer to use, instead of asynchronous I/O or an event loop. While POSIX defines an API for threads, called pthreads, several implementations are possible. The core of any threading implementation needs to provide two fundamentals, concurrency and synchronization. Concurrency allows a programm to accomplish multiple tasks, while providing the programmer with an abstraction that only one task need be addressed at a time. Synchronization permits multiple threads to interact in an orderly manner.

Userland Threads

One way to implement threads is entirely as a userspace library. The userlandapproach has two advantes. First, it works on operating systems which don't natively support threads. Second, for some tasks, it offers good performance. By not involving the kernel, syscall overhead is avoided.

By the same token, however, the kernel is unaware of the thread library's intentions. This means that it is subject to inoppurtune scheduling by the kernel. There's no true concurrency, but the only illusion of concurrency, achieved by replacing potentially blocking I/O calls with nonblocking calls. In practice, however, nonblocking I/O has a tendency to block, notably when reading from the filesystem. select() and poll() will always indicate that data is available, even when it isn't in the buffer cache. If one thread blocks waiting for data from disk, all threads in the same process block. This drawback severely handicaps the ability of any userland thread library to provide concurrency.

Kernel Threads and Scheduler Activations

To alleviate these shortcoming, support for threads was added to many systems' kernels. Now, the kernel can schedule another thread from the same process to run while one thread waits on disk, and a third thread can be running on a different CPU. A userland library still exists to provide the API, but many tasks, such as thread creation or synchronization, are delegated to the kernel through new syscalls.

The next stage of evolution for many thread implementations was a technique called Scheduler Activations. Originally developed by Anderson, et al, SA expands the userland/kernel thread interface to include a message passing system for all scheduler events. Instead of the kernel scheduler selecting a new thread to run when the currently running thread blocks, a message is sent to the library which then performs the task switch. SA were designed to improve the performace of operations like thread creation by avoiding a syscall, and increase the flexibility of the userland scheduler, by placing it in full control of thread scheduling.

OpenBSD and Threads

At the current time, the only supported thread model for OpenBSD is a userland library. It suffers from the typical set of problems. Anyone who has used a threaded media player on OpenBSD has likely discovered for themselves that when one thread blocks, they all block.

The introduction of SMP support for the i386 and amd64 architectures also highlighted the fact the because libpthread only utilizes one process, and therefore one scheduling entity, it could not take advantage of multiple CPUs. Several applications such as MySQL are written to utilize threads in an attempt to improve performance.

Rthreads

To address these issues, it was clear that kernel support for threads was required. Instead of an approach based on scheduler activations, implementations of which can be found in both FreeBSD and NetBSD, a direct 1:1 mapping of user threads to kernel threads was selected. The already existing rfork() system call provides a means to create multiple processes that share an address space - in effect, threads. In some ways, this is similar to the LinuxThreads library, particularly the FreeBSD port of which used rfork() as well. However, LinuxThreads relied on an extra control thread, and the kernel was unable to properly distinguish threads from processes.

Kernel Modifications

rfork() typically creates full fledged processes, not threads. Building a thread library directly on rfork() with no additional kernel support is possible; however, this leads to artifacts such as every thread appearing independently in the output of utilities such as ps and top. A new flag to rfork() was added, RFORK_THREAD, to indicate to the kernel that the new process should be considered a part of the parent. A linked list is maintained of threads for each process, similarly to the process sibling list. No separate thread structure has been created in the kernel. Threads are just processes with a special flag set.

All threads created so contain a thread parent pointer, which points to the struct proc for the process. The thread parent pointer for non-threaded processes is initialized to point back to itself. In this way, any access to data which particularly needs to address the process can be done through the thread parent pointer.

The advantage of this approach is that the kernel was made "thread-aware" with only changes to a few files - those dealing with process creation and exiting. When a thread of a process calls exit(), the kernel iterates over the list of sibling threads and also calls exit() for them. A new syscall was added to allow a single thread to exit. No other changes were initially necessary. As time goes on, more changes have been and will be made to more naturally integrate thread awareness into the kernel.

One disadvantage of the approach is that some of the struct proc fields are redundant for a thread. Future work will consider the feasibility of restructuring. In the mean time, the lossage even for thousands of threads measures only a few dozen KB.

In contrast to the schedular activation approach, the direct 1:1 mapping simplifies scheduling. Under SA, when a thread blocks in the kernel, a complex dance of interactions and upcalls is performed to find a new thread. This operation also occurs whenever a timeslice expires. Because an rthread is implemented as just another process in the kernel, nothing special need happen when it blocks. A new process is selected to run and the kernel performs its usual context switch operation.

Syscalls

The sys_rfork() system call is not new, but a modified version of sys_exit() needed to be provided so that one thread could exit by itself. The new syscall, sys_threxit(), simply calls exit1() with a special flag set to indicate only this thread intends to stop. Other threads may wait for an exiting thread using wait() like any other process. sys_getthrid() is the equivalent of sys_getpid(), although it does not map all threads to the parent process's pid. To support voluntary yielding, sys_yield() was added.

In order to support userland mutexes and semaphores, it was necessary to add two additional syscalls, sys_thrsleep(long ident, int timeout, void *lock) and sys_thrwakeup(long ident). These functions export the tsleep() and wakeup() kernel functions to userland. sys_thrsleep() is used to inform the kernel that the current thread wants to cease execution for an extended period (extended really only meaning more than a clock tick). The ident value is entered into a list of idents for the current process, and then tsleep() is called on the address of the list node. This enables the userland process to sleep on any address, much as a process on the kernel can block waiting on any address, while assuring that every process has a unique ident space and without requiring the kernel to interpret userland data. sys_thrwakeup() finds the node with the matching ident, then calls wakeup() on its address. A timeout may be specified to sys_thrsleep() to control the maximum sleep time. The final address is intended to be a spinlock currently held by the calling thread. The kernel will release just before calling tsleep(). It can be used to ensure that a second thread doesn't call sys_thrwakeup() before the first thread is fully asleep.

Library Code

Binary Compatible

The rthreads library is binary compatible with the pthreads library it replaces. The design of the original pthreads library was such that all exposed types are really pointers to opaque types. This means that compiled programs are agnostic to the size and organization of such types.

MD Code

The majority of rthreads code is machinde independent. On a per architecture basis, pieces of machine dependent code must be provided. The first is the rfork_thread(int flags, void *stack, void (*fn)(void *), void *arg) function. This function calls rfork(flags) and returns the thread id of the child to the parent. The child does not return. Instead, it has its stack pointer adjusted and jumps immediately to fn, passing it arg.

The second function is _atomic_lock(_spinlock_lock_t *lock) which performs an atomic compare and swap operation. This function is used to implement the userland spinlock functions.

Synchronization

pthreads includes several types of synchronization operations and data structures, such as simple mutexes, reader-writer locks, and condition variables. In rthreads, these are all implemented as layers on top of semaphores.

The semaphores are implemented using a combination of userland and kernel code. Spinlocks are used to protect the counter that indicates whether the semaphore is available. In the simple acquistion case, the count is adjusted and the spinlock released. In other cases, the thread must block until the semaphore becomes available. Blocking requires finding a new thread to schedule, so on rthreads a syscall is involved to inform the kernel. The blocking thread calls sys_thrsleep(), passing it both the address of the semaphore and the address of the semaphore's spinlock. The kernel then atomically releases the spinlock, and finds a new process to run or enters the idle loop.

The first thread will remain waiting on the kernel's wait list until a second process increases the semaphore count. If the current semaphore count is 0, the thread calls sys_thrwakeup().

Scheduling

One of the advantages often credited to SA is that the scheduling of threads is under control of the process and not subject to the kernel. Unfortunately, this flexibility comes at the cost of considerable complexity. At present, librthread has only limited control over the kernel scheduler. Ideally, some new syscalls can be added to expose more control to userland without undue complexity. Otherwise, it's possible for a running thread to yield the CPU at designated sequence points.

Future Work

Quite simply, signal handling is one the most complicated aspects of threads to get right. I'd also like to explore re-using kernel threads to improve performance, instead of calling threxit() immediately when finished. Some paradigms create a new thread to acomplish every small task and eliminating two or three syscalls will likely be a remarkable improvement.

Conclusion

The majority of the code and complexity with the old pthreads code dealt with trying to fake nonblocking I/O and scheduling. The requirement to perform the first has been eliminiated entirely, and the second task is now the responsiblity of the kernel. For this reason, rthreads is implemented using only a fraction of the amount of code previously required. rthreads is both a better and simpler replacement.

Thanks

Of course, any discussion of libpthread needs to mention John Birrel, its original author, and all the other FreeBSD developers who worked on it. All the OpenBSD developers, especially anyone who has worked on improving libpthread and who now face adapting many of those changes to librthread.

Bibliography

Anderson, et al. “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism”, *ACM Transactions on Computer Systems*, 1992.

Drepper, Ulrich and Ingo Molnar. “The Native POSIX Thread Library for Linux.”

Evans, Jason and Julian Elischer. “Kernel-Scheduled Entities for FreeBSD”

Williams, Nathan. “An Implementation of Scheduler Activations on the NetBSD Operating System”, USENIX 2002.