Reprinted from the

Proceedings of the Ottawa Linux Symposium

June 26th–29th, 2002
Ottawa, Ontario
Canada

Conference Organizers

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

Proceedings Formatting Team

John W. Lockhart, *Wild Open Source, Inc.*

POSIX Threads and the Linux Kernel

Dave McCracken
IBM® Linux® Technology Center
Austin, TX
dmccr@us.ibm.com

Abstract

POSIX® threading (commonly called pthreads) has long been an issue on Linux. There are significant differences in the multithread archictecture pthreads expects and the architecture provided by Linux clone().

This paper describes the environment expected by pthreads, how it differs from what Linux provides, and explores ways to add pthread compatibility to the Linux kernel without interfering with Linux's current multithread model.

1 Introduction

POSIX threads has become a widely used way of adding concurrency to an application. However, it doesn't map well onto Linux because of significant differences in how each of them defines a process, and the effects those differences have on the runtime environment.

In this paper we will describe the two models, how they differ, and offer some suggestions for changes to Linux that will allow it to emulate the POSIX model for those applications that use POSIX threads while preserving the current behavior for all other applications.

1.1 Definitions

In our discussion of POSIX threads, first we need to define some terms. Much confusion arises in thread discussions because of disagreement over what various terms mean. For the purposes of this paper we'll use the following definitions:

process Traditionally a UNIX® process corresponded to an instance of a running program. More precisely it was an address space and a group of resources all dedicated to running that program. This definition is formalized in POSIX. For this paper we will use the term 'process' to mean this POSIX definition.

thread The term thread comes from the concept of a single thread of execution, ie a linear path through the code. POSIX defines a thread to be the resources necessary to represent that single thread of execution. A process contains one or more threads. We will use the term thread in this paper when referring to the resources necessary to define a single execution path as seen by the application.

task In Linux, the basic unit is a task. In a program that only calls fork() and/or exec(), a Linux task is identical to a

POSIX process. The difference arises when a task uses the clone() system call to implement multithreading. The program then becomes a cooperating set of tasks which share some resources. We will use the term task to mean a Linux task.

2 History of POSIX Threads

Historically, the UNIX operating system has always had the concept of a process, which roughly equates to a running instance of a program. Each process has a set of resources associated with it, including an address space, a set of CPU registers, a process ID, a set of open file descriptors, a user ID, a stack, etc. While this is a powerful model, it only allows a single linear execution path. To gain any kind of concurrency with this model it is necessary to create multiple processes, often requiring some kind of inter-process communication, which is often expensive and unwieldy.

In the late 1980s, the concept of multiple threads of control became popular in the UNIX community. The fundamental idea was to take a limited set of a process's resources and make multiple instances of them, thus allowing concurrency within a single process. The resources selected were the minimum necessary to represent a single execution state. This primarily consisted of CPU registers and stack. Each instance was then called a 'thread'. This allowed concurrency within an application without the necessity of an inter-process communication mechanism. It is important to note that this model preserved the concept of a single process, and extended the definition to include multiple threads within that process.

At that time there were only a few experimental implementations of threads, and none in production. It was clear, however, that it addressed a growing need, especially with the prospect of multi-processor UNIX machines on the horizon.

At around this time the POSIX standardization effort was also underway. There was a strong push to create a common set of APIs that all UNIX implementations could be guaranteed to have. Several people put together an API that encapsulated the multi-thread model and proposed it for inclusion in POSIX. It was accepted in draft form under the real time extensions. While there was little real-world experience with threads at the time, the intent was to provide a common framework before multiple competing implementations appeared.

In the years since then the POSIX thread API (commonly known as pthreads) went through many revisions and was incorporated into the POSIX standard in 1996. Most if not all UNIX implementations include a pthread library, and there are many applications that use it.

3 POSIX Thread Model vs Linux Task Model

As we've stated before, the multithreading model used by POSIX is that of a single process that contains one or more threads. In contrast, the Linux multithread model is that of separate tasks that may share one or more resources. While this may sound like a small difference, the effects of this difference are far reaching.

3.1 Resources

The POSIX model is that all resources are global to the process except for the minimum set of resources that are necessary to represent a single thread of execution. This means that modifications to a resource will be seen by all other threads in the process.

In contrast, the Linux model has an independent set of tasks that have separate instances of all resources except for a few selected resources that may be shared. This sharing is selectable on a per-resource basis via flags passed to the clone() system call. All other resources have a separate instance for each task. A change to the resource in one task may not affect the equivalent resource in any other task.

The following resources are specific to a thread in POSIX, while all other resources are global to a process:

CPU registers
User stack
Blocked signal mask

The following resources may be shared between tasks via clone() in Linux, while all other resources are local to each task:

Address space
Signal handlers
Open files
Working directory

There are a number of resources that are process-wide in POSIX, but only task-specific in Linux, that cause compatibility problems. A partial list of the ones that cause the most problems includes process ID, parent process ID, credentials (user ID, group ID, etc), and pending signal mask.

3.2 Process-wide Actions

The fundamental difference between the models is that in POSIX a process can be addressed as a single entity, while in Linux it is a collection of independent tasks. There are several actions that can be done both from outside the process and within the process that will affect the whole process. In Linux, however, each of these actions will only affect one task, leaving the other tasks to continue without knowledge of the event.

The actions that are of special concern are:

Signals POSIX states that all signals sent to a process will be collected into a process-wide set of pending signals, then delivered to any thread that is not blocking that signal. Linux only supports signals that are sent to a specific task. If that task has blocked that particular signal, it may remain pending indefinitely.

Exit In POSIX, there are several actions that can request the death of the entire process. All threads are killed and the process exits with a status indicating the cause of the death. All of these actions in Linux will only kill the specific task, leaving all other tasks unaffected.

Suspend/Resume Certain signals have the default action of doing a suspend or resume. In POSIX, this action is defined to take effect on the entire process, which is translated to include all threads in that process. In Linux, the action only takes effect on the task the signal was delivered to.

Exec In POSIX, the effect of the execve() system call is to terminate all threads in the process, throw away the address space, and instantiate a new address space with a single thread. In Linux, if there is more than one task that shares the address space, the task that calls execve() is detached from the address space and has a new one created. All other tasks sharing that address space will continue to run.

4 Implementations of Multithread Libraries

4.1 Threading Styles

Multithread libraries typically come in one of three basic styles. Each has its advantages and disadvantages.

4.1.1 M:1

The first style, M:1, implements all threads in user space and appears to the kernel as s single-threaded process. This style is the most portable, in that it does not require any special features from the underlying kernel. One drawback is that it requires that all blocking system calls be emulated in the library via non-blocking calls to the kernel. This emulation adds significant overhead to system calls, in particular most IO. There are also some blocking system calls that can not be emulated via non-blocking calls. When these calls are used the entire process blocks. This style also does not allow the application to take advantage of any multiprocessor scheduling, since to the kernel scheduler it is still a single-threaded process.

This style is primarily of historical interest. Most current operating systems provide some support for multithreading at the kernel level, which provides improved performance.

4.1.2 1:1

The second style, 1:1, creates a kernel thread or task for each application thread. This has the advantage of being the simplest to implement at the library level, but each thread created becomes more expensive of kernel resources. This style is also the most dependent on the multithreading model of the underlying kernel.

There are some types of applications where this style is desirable, primarily when the application wants to create a small number of threads that each act independently and spend much of their time in runnable state.

4.1.3 M:N

The third style, M:N, provides the most flexibility. It is like M:1 threading in that it does not create a kernel thread for each application thread. The library creates multiple kernel threads, then schedules application threads on top of them. Most M:N thread libraries will dynamically allocate as many kernel threads as it needs to service the application threads that are actually runnable. This style is in some ways more heavyweight in that scheduling is occurring both in the kernel among the kernel threads for the process and in the library for the application threads, but it has the advantage of not consuming kernel resources for the application threads that are not actually runnable. This style also provides significantly better performance when threads in an application are synchronizing with each other, ie taking local mutexes. The library-level scheduler can switch between threads much faster because it doesn't have to enter the kernel.

Most multithreaded application perform better with this style, particularly applications that create large numbers of threads that only run sporadically.

4.2 Current Linux Thread Libraries

There have been multiple efforts to provide a pthread-compliant library for Linux. Early on in Linux's history only M:1 thread libraries were created, but were mostly abandoned as Linux developed better multithreading support at the kernel level.

The default library shipped with all the distributions is currently LinuxThreads. It is supported by the same group that provides glibc. The LinuxThreads library provides a pthread API, but internally it is primarily a wrapper for the Linux task model. It uses the 1:1 style, creating a task for each application thread using clone() and sharing the address space, the signal handlers, and the open files. This approach generally performs well, but the underlying differences from the POSIX thread model are exposed to the application. Applications that were coded to work with pthreads as specified by the standard may not work, and must be ported.

There is a new pthread library under development called NGPT. This library is based on the GNU Pth library, which is an M:1 library. NGPT extends Pth by using multiple Linux tasks, thus creating an M:N library. It attempts to preserve Pth's pthread compatibility while also using multiple Linux tasks for concurrency, but this effort is hampered by the underlying differences in the Linux threading model. The NGPT library at present uses non-blocking wrappers around blocking system calls to avoid blocking in the kernel.

5 Linux Kernel Changes for POSIX Compatibility

While it would be possible to emulate POSIX compatibility in a library, it would be extremely painful in many areas. A much simpler solution would be to add compatibility code to the Linux kernel, either to provide compatible behavior or provide hooks that would make it easier for a library to provide it. In this section we will describe some changes that make POSIX compatibility feasible. Some have already been included, some have patches available, and some have not yet been addressed. All the changes are intended to be optional, only enabled by request from the application or library. This would most likely be via additional flags to the clone() system call.

5.1 Thread Groups

One of the fundamental barriers to adding POSIX compatibility to Linux has been that Linux had no easy way to group all the tasks together that are part of what POSIX would call a process, and iterate through them. It was possible to find all tasks with the same address space, but only by looking at all tasks in the system. This limited what could be added at the kernel level.

This was addressed during the 2.4 development cycle with the addition of a concept called a 'thread group'. There is a linked list of all tasks that are part of the thread group, and there is an ID that represents the group, called the tgid. This ID is actually the pid of the first task in the group (pid is the task ID assigned with a Linux task), similar to the way sessions and process groups work. This feature is enabled via a flag to clone().

The task whose ID becomes the tgid is known as the ’thread group leader’. This task takes on special properties, since in most library implementations it will be the initial task running after exec(), and its ID is the one known to the parent who originally invoked the application.

As part of the thread group change, the getpid() system call was changed to return tgid instead of pid. This means that all tasks in a thread group will see the same pid. While this is correct for applications, pthread libraries will still need to be able to get the actual pid of the task, so the gettid() system call was added for them.

A corollary to the getpid() system call is getppid(). At present it returns the pid of the task that cloned() the task making the system call. For POSIX compatibility it should return the parent ID of the thread group leader.

While thread groups by itself only adds limited functionality, it provides the grouping necessary for other changes that will improve compatibility.

5.2 Signals

Signals have long been a difficult issue, beginning with early versions of the UNIX system. The question of how to handle signals in a multithreaded process has been debated since the early days of POSIX threading, and went through extensive changes in various drafts of the standard.

The kernel state maintained for a given signal consists of three pieces of information, the signal handler, the blocked flag, and the pending flag. The signal handler is an address of a user-level function to run when the signal is received. Special values of the signal handler allow the application to specify default behavior for that signal or to ignore it completely. The blocked flag is a flag that can be set by the application to temporarily prevent the signal from being delivered. The pending flag is set whenever that signal is sent to the application, and reset when the signal is actually delivered, ie the handler is run or other action is taken.

Signal handlers in Linux can be either per-process or per-task, controlled by a flag to clone(). This allows POSIX compatibility for handlers. POSIX specifies that the blocked flag should be per-thread, so the existing Linux behavior of having blocked flags for each task is compatible with POSIX.

The compatibility issue arises with the pending signal flag. POSIX states that signals are sent to the entire process, which means a thread context must be selected to run the handler. POSIX specifies that the delivery code must search the threads in the process and find one that does not have that signal blocked. If all threads are blocking that signal, it remains pending until one thread unblocks it, at which time that thread will run the handler. If more than one thread is not blocking the signal, POSIX does not specify which one will run the handler.

In Linux, all signals are sent to a specific task. Each task has its own pending signal flags, and the flag for that signal will be set. If that task has that signal blocked, it will remain pending until the task unblocks it, even though there may be other tasks in the process that do not have it blocked.

It is possible to partially emulate POSIX behavior in a pthread library by providing a complete signal layer, complete with its own handler array, per-thread blocked masks, and pending signal mask. This requires that the library register its own signal handler in the kernel for all signals, and to not block signals at the kernel level. The biggest problem with this approach is the significant added complexity and performance cost of duplicating the functionality. There are also circumstances where the application will still see interrupted system calls when all threads are blocking a signal or the signal is supposed to be ignored.

In support of the NGPT project I wrote a patch that allows libraries to provide POSIX signal emulation. The patch works in conjunction with thread groups. When a signal arrives for any task in a thread group, that signal is redirected to the thread group leader. This allows a pthread library to leave signals unblocked in the thread group leader task, and receive all signals directed at any task in the process. It doest not directly support POSIX compatibility, but gives the library the tool it needs to provide its own compatibility.

The thread group leader patch has some drawbacks of its own, however. It creates a bottleneck in an application with large numbers of signals. It also still requires significant code in the library to handle blocking signals for each thread, ie if all threads block a signal, it still needs to be blocked at the kernel level.

Another issue with this approach is that it would make it more difficult to do a thin 1:1 pthread library, since it would still have to provide significant signal code in the library. A better solution for this would be to actually add a shareable structure to the kernel for pending signals, with the attendant code to check all tasks in a thread group to see whether any of them can receive the signal. This solution would also address the bottleneck issue.

5.3 Credentials

Credentials are the collective identity associated with a process or task, ie the user ID, the group ID, the list of groups, and the capabilities. POSIX states that the credentials are per-process, ie when one thread within the process changes some part of the credentials, all threads see the change. In Linux, the credentials are per-task, so it's possible to have two tasks in a process running under different user IDs, for example.

The simple solution to this is to change credentials to be a shareable structure. This would preserve existing behavior, but allow processes that wish POSIX behavior to share credentials.

5.4 Semaphore Undo

Another resource that under POSIX is process-wide is System V semaphores. This primarily becomes an issue when an application uses the undo feature. This feature will reset semaphores on process exit. In Linux, the semaphore state is per-task, so when each task exits it will undo the semaphore. POSIX processes assume that the semaphore will continue to maintain its state until the entire process exits.

This problem is another one that can be solved by sharing state between tasks when a flag is passed to clone(). A patch for this exists, but has not yet been accepted.

5.5 Process-wide Actions

There are some actions that POSIX defines to be process-wide which under Linux are per-task. Some of these actions are initiated from inside the kernel and can not be detected and emulated inside a library.

Exit A difficult compatibility issue is that of exit. POSIX defines several actions that can result in the entire process exiting, including the exit() system call and default actions for many signals. This process exit should produce an exit status that can be passed to a waiting parent. This means that any thread in the process can cause the entire process to exit and produce a status back to a waiting parent.

The Linux behavior is dramatically different. Each of these exit actions results in the termination of a single task, leaving all other tasks in the process running. If the task is the initial one created by fork(), the parent will receive its exit status and may assume the process has exited when in fact it is still running in other tasks.

Exec Under POSIX, an execve() system call from any thread in a multithreaded process will cause all other threads in that process to terminate and the calling thread will complete the exec. The entire address space associated with that process will be discarded, and a new one created.

In Linux, when a task calls execve(), it is detached from the address space, then a new address is created to complete the exec. If any other task is using the old address space it will continue to run.

Suspend/Resume Some signals have the default action of initiating a suspend or a resume. POSIX states that this will occur on the entire process by suspending or resuming all threads in that process. Linux only applies the suspend or resume to the task receiving the signal and does not affect any other task.

A possible solution for these would be a kernel function that iterates through an entire thread group and applies the requested action to each task in that group. Special care would have to be taken to preserve the proper exit status to any waiting parent. Synchronizing all the tasks in a thread group is expected to be a difficult problem.

6 Conclusion

We have shown how POSIX threading uses a different model than the Linux task model, and how that affects pthread libraries on Linux. We have also discussed some things that have been and could be done to the Linux kernel to better allow pthread libraries to emulate the POSIX behavior. These changes could be added without disrupting the current Linux task behavior, allowing Linux to support both the POSIX multithread model and its own cooperating task model.

Lawyer Foo

This paper represents the views of the author, and not the IBM Corporation.

IBM® is a registered trademark of International Business Machines Corporation.

UNIX® is a registered trademark of The Open Group.

POSIX® is a registered trademark of the IEEE.

Linux® is a registered trademark of Linus Torvalds.

Other company, product or service names may be the trademarks or service marks of others.