IBM

Linux threading models compared: LinuxThreads and NPTL

A rundown of the key differences for developers who need to port

Level: Intermediate

Vikram Shukla (vikshukl@in.ibm.com), Software Engineer, IBM

31 Jul 2006

The LinuxThreads project originally brought multithreading to Linux®, but LinuxThreads didn't conform to POSIX threading standards. Although the more recent Native POSIX Thread Library (NPTL) library has filled in some of the gaps, other issues remain. This article describes some of the differences between these two Linux threading models for developers who may need to port their applications from LinuxThreads to NPTL or who simply want to understand where the differences lie.

When Linux was first developed, it did not have real support for threading in the kernel. But it did support processes as schedulable entities through the `clone()` system call. This call created a copy of the calling process, with the copy sharing the address space of the caller. The LinuxThreads project used this system call to simulate thread support entirely in user space. Unfortunately, this approach had a number of disadvantages, particularly in the areas of signal handling, scheduling, and interprocess synchronization primitives. Moreover, the threading model did not conform to POSIX requirements.

To improve on LinuxThreads, it was clear that some kernel support and a rewritten threads library would be required. Two competing projects were started to address these requirements. A team including developers from IBM worked on NGPT, or Next-Generation POSIX Threads. Meanwhile, developers at Red Hat were working on the NPTL. NGPT was abandoned in mid-2003, leaving the field to NPTL.

Although the choice of NPTL over LinuxThreads seems like a foregone conclusion, if you're maintaining apps for an aging Linux distribution and plan to upgrade soon, migrating to NPTL will be an important part of the porting process. Alternatively, you may want to know about these differences so you can design your applications to accommodate both the older and the newer technologies.

This article details which threading models are implemented on which distributions.

LinuxThreads design specifics

Threads split a program into one or more simultaneously running tasks. Threads differ from the *processes* of traditional multitasking in that threads share the state information of a single process and share memory and other resources directly. Context switching between threads in the same process is typically faster than context switching between processes. Hence, the advantage of a multithreaded program is that it can operate faster than a multiprocessed application. Furthermore, with threads you can implement simultaneous processing. These relative advantages over a process-based approach led to implementation of LinuxThreads.

The initial design of LinuxThreads started with the belief that context switches among related processes are fast enough that each kernel thread can handle a corresponding user-level thread. This led to the evolution of the *one-on-one* threading model.

Let's review the high points of LinuxThreads' design specifics:

- One of the notable features of LinuxThreads is the *manager thread.* The manager thread fulfilled these requirements:

  - The system must be able to react to fatal signals and kill the entire process.
  - The de-allocation of memory used as stacks must happen after a thread is finished. Therefore, the thread cannot do this itself.
  - Terminating threads must be waited on so that they don't turn into zombies.
  - The de-allocation of thread-local data requires iterating over all threads; this must be done by the manager thread.
  - If the main thread needs to call `pthread_exit()`, the process is not terminated. The main thread goes to sleep, and it is the job of the manager thread to wake up the main thread when all other threads have been killed.

- To maintain the thread-local data and memory, LinuxThreads uses the top memory of the process address space just below the stack address.

- The synchronization of primitives is achieved by means of *signals.* For example, threads block until awoken by signals.

- Under the initial design of the clone system, LinuxThreads implemented each thread as a different process with a unique process ID.

- A fatal signal is able to kill all the threads. The LinuxThreads design on this front has been consistent. Once a process receives a fatal signal, the thread manager kills all the other threads (processes) with the same signal.

- According to LinuxThreads design, if an asynchronous signal is sent, the manager thread will deliver the signal to one of threads. If that thread is currently blocking the signal, the signal remains pending. This is because the manager thread cannot send a signal to a process; instead, each thread acts a process.

- Scheduling between threads is handled by the kernel scheduler.

LinuxThreads and its limitations

The design of LinuxThreads worked fine in general; but, when stressed by intensive applications, it suffered problems with performance, scalability, and usability. Let's look at some of the limitations of the LinuxThreads design:

- It uses the manager thread to create and coordinate among all threads owned by each process. This increases the overhead of creating and destroying threads.

- Because it's designed around a manager thread, it causes a lot of context switching, which potentially hampers scalability and performance.

- Because the manager thread can run on only one CPU, any synchronization performed can cause scalability problems on SMP or NUMA systems.

- Because of the way that threads are managed, and because each thread has a different process ID, LinuxThreads is incompatible with other POSIX-related threading libraries.

- Signals were used to implement synchronization primitives. This affected the response time of the operations. Moreover, the concept of sending a signal to the main process as such is not present. This, therefore, does not conform to the POSIX way of handling signals.

- Point signal handling within LinuxThreads is conducted on a per-thread basis rather than on a per-process basis as each thread has a separate process ID. Since a signal is sent to a dedicated thread, signals are *serialized* -- that is, the signals are funneled through this thread to other threads. This is in contrast to the POSIX standard's

requirements for parallel handling of signals. For example, under LinuxThreads, signals sent via `kill()` are delivered to individual threads rather than to the process as a whole. This means that if that thread is blocking the signal, then LinuxThreads will simply queue in that thread and execute the handler only when that thread unblocks the signal, instead of executing the handler immediately in the other thread that does not block the signal.

- Since each thread in LinuxThreads is a process, it's possible that user and group ID information will not be common to all threads in a single process. So, for example, a multithreaded `setuid()`/`setgid()` process could be different for different threads.

- There are instances where the multithread core dump created does not contain all the thread information. Again, this behavior is a result of the fact that each thread is a process. If a crash happens on any of the threads, we see only that thread on a system core file. However, this behavior applies mainly to on older versions of the LinuxThreads implementation.

- Because each thread is a separate process, the /proc directory is flooded with lots of process entries that ideally should have been threads.

- Because each thread is a process, there is a limited number of threads that can be created for an application. For example, on an IA32 system, the total number of processes possible -- and thus, the total number of threads that can be created -- is 4,090.

- Because the method of calculating the thread-local data was based on the position of the stack address, access to that data was very slow. Another disadvantage was that the user could not specify the size of the stack confidently, as the user could accidentally map the stack address to an area that was meant to be used for different purpose. The *grow on demand* concept (also called the *floating stack* concept) was implemented in version 2.4.10 of the Linux kernel onwards. Prior to this, LinuxThreads used a fixed stack.

About NPTL

NPTL, or Native POSIX Thread Library, is a new implementation of Linux threading that overcomes the disadvantages of LinuxThreads and also conforms to POSIX requirements. It offers significant improvements over LinuxThreads in terms of performance and stability. NPTL, like LinuxThreads, implements the one-on-one model.

Ulrich Drepper and Ingo Molnar are two employees of Red Hat who participated in NPTL's design. Some of their overall design goals were as follows:

- The new threading library should be POSIX compliant.

- The thread implementation should work well on systems with large number of processors.

- Creating new threads for even a small piece of work should have a low startup cost.

- The NPTL thread library should be binary compatible with LinuxThreads. Note that you can use `LD_ASSUME_KERNEL`, which is discussed later in this article, for this purpose.

- The new threading library should be able to take advantage of NUMA support.

Advantages of NPTL

NPTL holds a number of advantages over LinuxThreads:

- NPTL does not use a manager thread. Some of the requirements of the manager thread, like sending fatal signals to all the threads that were part of the process, were not required, as the kernel itself can take care of them. The kernel also takes care of de-allocating the memory used by each thread stack. It even manages the termination of all the threads by waiting on them before clearing the parent thread, thus avoiding zombies.

- Because it doesn't use a manager thread, the NPTL threading model has better scalability and synchronization mechanisms on NUMA and SMP systems.

- With NPTL threading libraries along with the new kernel implementation, the synchronization of threads by means of signals was avoided. For this purpose, NPTL introduces a new mechanism called a *futex*. A futex works on shared memory regions and hence can be shared between processes, thus providing interprocess POSIX synchronization. It's also possible to share a futex across processes. This behavior made interprocess synchronization a reality. In fact, NPTL includes a macro called `PTHREAD_PROCESS_SHARED` that gives a handle to the developer to make a user-level process share a mutex across threads of different processes.

- Since NPTL is POSIX compliant, it handles signals on a per-process basis; `getpid()` returns the same process ID for all the threads. For example, if a signal `SIGSTOP` is sent, the whole process would stop; with LinuxThreads, the thread that received the signal would stop. This enables better usage of debuggers like GDB on NPTL-based applications.

- Since in NPTL all threads have one parent process, the resource usages reported to the parent (like CPU and memory percentages) are reported for the entire process, not for just one thread.

- One of the important features that was introduced by the NPTL threading library was support for ABI (Application Binary Interface). This helped enable backward compatibility with LinuxThreads. This can be done with the help of `LD_ASSUME_KERNEL`, which is covered next.

The LD_ASSUME_KERNEL environment variable

As explained above, the introduction of ABI made it possible for code to support both the NPTL and LinuxThreads models. Basically this is taken care by ld (a dynamic linker/loader), which decides which runtime threading library to dynamically link in.

As an example, here are some common settings for this variable that WebSphere® Application Server uses; try these as appropriate for your requirements:

- `LD_ASSUME_KERNEL=2.4.19`: This overrides the NPTL implementation. This implementation is commonly referred to as the standard LinuxThreads model with floating stacks feature enabled.
- `LD_ASSUME_KERNEL=2.2.5:` This overrides the NPTL implementation. This implementation is commonly referred to as LinuxThreads with fixed stack size.

Set this variable with following command:

```
export LD_ASSUME_KERNEL=2.4.19
```

Note that support for any `LD_ASSUME_KERNEL` setting will depend on the ABI versions currently supported for the threading library. For example, if any threading library does not support ABI version 2.2.5, then the user will not be able to set `LD_ASSUME_KERNEL` to 2.2.5. Typically, NPTL requires 2.4.20, and LinuxThreads requires 2.4.1.

All these settings generally come into use if you're running on an NPTL-enabled Linux distribution but your application has been designed on the basis of the LinuxThreads model.

The GNU_LIBPTHREAD_VERSION macro

Most modern Linux distributions ship with both LinuxThreads and NPTL, and they provide a facility to switch between the two. To discover which version of which thread library you're currently using on your system, run the following command:

```
$ getconf GNU_LIBPTHREAD_VERSION
```

The output will look something like this:

```
NPTL 0.34
```

Or:

```
linuxthreads-0.10
```

Linux distributions with threading model, glibc version, and kernel versions

Table 1 lists some of the popular Linux distributions, along with the type of thread implementations, glibc libraries, and kernel version for each.

Table 1. Linux distributions and their threading implementations

Threading implementation	C library	Distribution	Kernel
LinuxThreads 0.7, 0.71 (for libc5)	libc 5.x	Red Hat 4.2	
LinuxThreads 0.7, 0.71 (for glibc 2)	glibc 2.0.x	Red Hat 5.x	
LinuxThreads 0.8	glibc 2.1.1	Red Hat 6.0	
LinuxThreads 0.8	glibc 2.1.2	Red Hat 6.1 and 6.2	
LinuxThreads 0.9		Red Hat 7.2	2.4.7
LinuxThreads 0.9	glibc 2.2.4	Red Hat 2.1 AS	2.4.9
LinuxThreads 0.10	glibc 2.2.93	Red Hat 8.0	2.4.18
NPTL 0.6	glibc 2.3	Red Hat 9.0	2.4.20
NPTL 0.61	glibc 2.3.2	Red Hat 3.0 EL	2.4.21
NPTL 2.3.4	glibc 2.3.4	Red Hat 4.0	2.6.9
LinuxThreads 0.9	glibc 2.2	SUSE Linux Enterprise Server 7.1	2.4.18
LinuxThreads 0.9	glibc 2.2.5	SUSE Linux Enterprise Server 8	2.4.21
LinuxThreads 0.9	glibc 2.2.5	United Linux	2.4.21
NPTL 2.3.5	glibc 2.3.3	SUSE Linux Enterprise Server 9	2.6.5

Note that from kernel 2.6.x and glibc 2.3.3 onwards, the version numbering convention for NPTL seems to have changed: the library is now numbered in accordance with the glibc version being used.

Java™ Virtual Machine (JVM) support can vary. IBM's port of the JVM supports most of the distributions in Table 1 that have glibc versions greater than 2.1.

Conclusion

The limitations of LinuxThreads have been overcome in NPTL, as well as in some later versions of LinuxThreads. For example, the latest LinuxThreads implementation uses thread registers for locating thread-local data; on Intel® processors, for instance, it uses the `%fs` and `%gs` segment registers to locate the virtual address to access thread-local data. Though the results have shown improvement with the changes that have gone into LinuxThreads, problems still resurface under higher loads or stress tests because of factors like overdependence on the manager thread, issues with signal handling, and so on.

You should also keep in mind that while building your library with LinuxThreads, use the `-D_REENTRANT` compile-time flag. This makes the library thread safe.

Finally, and perhaps most importantly, remember that LinuxThreads is no longer being actively updated by the originators of the project, who see NPTL as a replacement.

The drawbacks of LinuxThreads should not be taken to mean that NPTL is error free. NPTL, being an SMP-oriented design, has drawbacks too. I have seen cases on recent Red Hat kernels where a simple threading application runs fine on a single-processor machine but hangs on SMP. I believe there is still more work to be done on Linux to truly make it scalable to satisfy higher-end applications.

Resources

Learn

- "The Native POSIX Thread Library for Linux" (PDF), by Ulrich Drepper and Ingo Molnar, describes the reasons and goals for designing NPTL, including the shortcomings of LinuxThreads and advantages of NPTL.
- The LinuxThreads FAQ covers frequently asked questions on LinuxThreads and NPTL. It's a useful resource for learning a few of the shortcomings in older implementations of LinuxThreads.
- "Explaining LD_ASSUME_KERNEL," by Ulrich Drepper, provides details about this environment variable.
- "Native POSIX Threading Library (NPTL) support describes the difference between LinuxThreads and NPTL from the WebSphere perspective and explains how WebSphere Application Server supports these two different threading models.
- Diagnosis documentation for IBM ports of the JVM defines the diagnostic information to be collected when a Java application faces issues while running on Linux.
- In the developerWorks Linux zone, find more resources for Linux developers.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- The LinuxThreads README gives a general description of LinuxThreads.
- Order the SEK for Linux, a two-DVD set containing the latest IBM trial software for Linux from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

Discuss

- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Vikram Shukla, with more than six year's experience in development and design using object-oriented languages, currently works as a staff software engineer in the Java Technology Center at IBM, Banglore, India, supporting IBM JVM on Linux.