Processes and Threads – Background and Current Developments

Clone Wars

Thread handling is critical to the performance and parallelization potential of a Linux application. The following article describes how threads and processes work, and looks into current developments in this field including the differing libraries used to implement under Linux and other Unixes.

BY WOLFGANG HETZLER AND ULRICH WOLF

Peter Doeberl, visipix.com

All modern operating systems use pre-emptive multitasking to distribute a large number of activities evenly across the processing resources. There are two possible approaches: processes and threads. Processes are the traditional approach, threads the more modern variant. Each UNIX version handles processes the same way, but when it comes to thread handling, a large number of different implementations become apparent.

Threads can simplify the development of applications for parallel computing; many critical applications that started life on other operating systems make heavy use of threading. Programming languages such as Java are even based on threads. This is why the subject cropped up again within the Linux context just a few years ago.

It is hardly surprising that hardware manufacturers have taken the initiative and actively encourage development. After all, it is in their own interests to be able to rebuff the claim heard in some quarters that "Linux does not scale".

The battle for the best possible solution for Linux is still currently raging; we will be looking at the three most likely candidates for our future systems. Like it or not, we will need to look at some theory before we start.

Heavyweights and Athletes

The concept of processes is one of the most elementary aspects of UNIX. Older operating systems used similar concepts, such as tasks or jobs – a unit of work that could be assigned resources by the operating system. The operating system guru, Professor Andrew Tanenbaum, sums this up: a process is "A program being executed".

The implementation of processes on Linux differs only slightly from the classical UNIX process. The kernel uses scheduling to define how long a process can utilize a system's resources. However, scheduling algorithms can be different depending on whether they are used with Linux or UNIX.

The only way to create a new process is the system call to "fork()" (or its older variant "vfork()"), and this assumes that a parent process is already running.

At first, the new process is a perfect copy of the original, and this includes its memory image, all its variables and registers. Thus, the child process now possesses a complete copy of the address space used by the parent process – there is no such thing as memory sharing, with the exception of things like shared memory of course.

The process ID is used to distinguish between the parent and child process. A system call to "fork()" returns a positive integer to the parent process and "0" to the child process.

In practical applications, the complete copying of memory content on forking is too complex, as it requires a series of consecutive memory manipulation operations. This is why all modern UNIX versions and Linux use a technique called copy on write.

The child process still receives a few page tables, however, these point to the pages of the parent and are write protected for the child process. It is not until

one of these processes attempts to write to a page that copying really takes place.

The system call to "fork" occurs via a function called "do_fork()"; the essential element for storing information on processes is a structure called "task_struct"; this is referred to as the process descriptor. All process descriptors are linked by means of a double linked list. Process data and its execution are not separate entities. In contrast to this, threads expect to be passed a process as a data quantity, but spawn a separate entity to execute a task.

Thus, a process can comprise multiple threads, which are referred to as light weight processes. The threads are a special type of process and are also referred to as Light Weight Processes (LWP), which are an extension of the UNIX standard.

Linux uses a system call to "clone()" ("man 2 clone") which is not available in the Posix based UNIX standard. The call works in a similar way to "fork", apart from the fact that "clone()" allows the child "process" to share resources with the calling process. Programmers can use a bitmap, or so-called "sharing_flags", to specify what the child process inherits. In earlier Linux versions this bitmap comprised five bits, but it has grown to 17 in version 2.5.

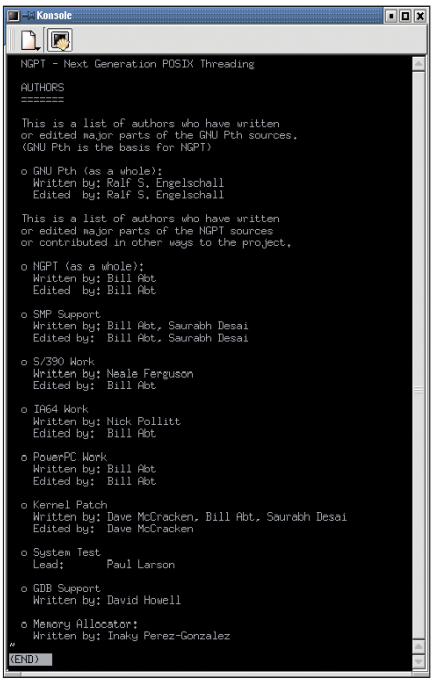


Figure 1: The "AUTHORS" file in NGPT shows that non-Intel systems, at least those produced by IBM, were important to the developers

The most important bit is "CLONE_VM"; if this bit is set, the new process will share the address space of the original process, thus producing an LWP, that is a thread. The "CLONE_FS" controls whether directories and the "umask" are shared. If the "CLONE_FILES" bit is set, file descriptors will be shared, and "CLONE_SIGHAND" does the same for the signal handler.

"CLONE_PID" is a bit more tricky; you might assume that this would allow the programmer to use the PID of the parent process to create the child process. This is not true. Only a process with a PID of "0", or so-called "idlestasks", are allowed to set this bit. Multiprocessor systems provide an "idlestask" for each CPU. The 2.5 kernel renames "CLONE_PID" to "CLONE_IDLETASK " to remove any confusion.

Special threads that are typically referred to as service threads, but occasionally called kernel threads, cause some confusion. The problem is that the threads we have referred to so far are implemented at kernel level and thus often known as kernel threads. The special type, that is the service thread, spends its whole life in kernel space – in contrast to normal processes – and are thus normally used to perform a few specific functions. The process we referred to previously, with a PID of "0" is a kernel thread of this type. Kernel threads are created by the "kernel_thread()" function which uses "do_fork()" just like "clone()".

Posix Threads – the Common Denominator

Threads offer a whole range of advantages in contrast to the more unwieldy processes. Shared resources offer better performance than would be possible if the same task had to be performed by multiple processes; complex interprocess communication, using pipes for example, is no longer needed. The threads also allow for better utilization of multiprocessor systems, as the threads of a single process can be distributed across several CPUs.

Of course there are disadvantages – synchronizing threads is extremely complex. Finally, anything that uses "clone" is specific to Linux and not portable. As this problem is not new, but occurred previously in the history of UNIX, it is reflected in the Posix standardization approach, in the IEEE 1003.1c standard to be more precise, where the definition of Posix threads are laid down.

Threads within the context of libraries

Posix threads are defined for userspace only. The specifications does not describe how the kernel is supposed to handle them. The obvious answer is to write thread libraries that abstract from the underlying system and thus provide Posix compatibility.

Library functions of this kind provide generic handling – specifics required by the current UNIX system then become internals. These functions map user-space threads to kernel threads or processes. The way this is performed is one of the most important criteria on which to judge a thread library. This is often a confessional issue and not exactly new. The following relationships are possible: 1:n, m:n, and 1:1.

The 1:n approach refers to the strategy which is typically most useful for the UNIX systems that do not provide a suitable thread handling strategy of their own. Threads remain at the user level of a process that is responsible for handling. The kernel will only see this one process. The m:n approach is a jack of all trades as it uses kernel level thread implementation on one side, and pro-

A Selection of Threading Libraries

Library	Since	Implemented in	Mapping Model
NGPT	2001	User/Kernel	n:m
NPTL	2002	User/Kernel	1:1
Gnu Pth	1999	User	1:n
Linuxthreads	1997	User/Kernel	1:1
Uthread	1998	User	1:n
Solaris Pthread	1991	User/Kernel	n:m, later 1:1
AIX Pthreads	1993	User/Kernel	1:1

vides additional library functions for management purposes. Scheduling thus takes place both at kernel and function level. At user level m threads are handled by n threads at kernel level.

The advantage of this complex approach is that it allows multiprocessor machines in particular to maximize system load. Older thread libraries, such as Solaris and Tru64 use this approach. Developers can decide for themselves how to map to kernel threads; in practical applications that number of kernel threads tend to be an integer which is a multiple of the number of CPUs.

A 1:1 relationship is easier to implement. The kernel, which supports threading, assumes complete responsibility for scheduling. The library is used to ensure compatibility. Most thread libraries adopt a 1:1 approach.

Since the mid 90's there have been several thread libraries, although only Linuxthreads by Xavier Leroy, which is implemented in the Pthread lib has been able to make a lasting impression. While developing Glibc2, Ulrich Drepper tied threading functionality closely to the standard library; thus threads have been available on every Linux system for quite a while now.

Current Thread Libraries

In its Linux implementation, the Pthread library uses a modified 1:1 relationship, where at least one main thread is produced; however "clone()" is used to create all threads. The more interesting Linux became for server farms and multiprocessor systems, the more vociferous the critics of this thread library became..

Unsatisfactory signal handling, hierarchical relations between threads instead of peering, imperfectly implemented Posix compatibility and the fact that the Pthread library threads – despite theoretical objections – are visible in "/proc", were just a few of the criticisms levied.

Recently, three particularly active projects have emerged from a whole range of approaches. Ralf S. Engelschall launched Gnu Pth [3] way back in 1999, and this project is now approaching the transition from 1.4x to 2.0. Although this is an official GNU project, the latest betas and information are only available from Ralf Engelschall's own website *www.ossp.org*, and on Freshmeat. Gnu Pth is designed to be a portable library for non-pre-emptive multitasking. It offers a Posix compatibility mode and maps userspace threads 1:n. The author describes the theory this is based on in great depth [4].

Five Intel and IBM programmers used Gnu Pth as a starting point for developing Next Generation Posix Threads (NGPT). As their employer placed a great deal of emphasis on enhancing scalability for multiprocessor systems, this project uses an m:n mapping model.

The current version, 2.2.0, was released in January 2003 and is described as stable by IBM. There was no documentation available when this issue went to print, the source packages contained only the documentation for the original Gnu Pth project. However, the website does give some hope with the promise of white papers and manpages. To test NGPT, all you need is a 2.4.19 kernel, although it will need patching. An installation guide is available from [5].

Red Hat to the fore

The NPTL (Native Posix Thread Library) by Ulrich Drepper and Ingo Molnar – both of whom work for Red Hat – is again based on the 1:1 model, although it does do without a main or manager thread, instead resorting to kernel modifications (new system calls that support threads, an enhanced clone call, modification of PIDs and signal handling, thread local storage). All of this will be available in Red Hat 8.1.

Drepper and Molnar are currently working overtime on a new library, new minor releases of the 0.x series appear every week or at even shorter intervals [6], [7]. NPTL currently requires a developer kernel from the 2.5 series, a current Glibc2.3 (whose maintainer just happens to be Ulrich Drepper). An installation guide is available on the Web (although this may already be obsolete by now) [8]. Binary packages, which have been down ported to 2.4 kernel, are included with Red Hat's 8.1 Beta (Phoebe). Both of Red Hat's top developers are gunning to make their thread library the future Linux standard.

There are benchmarks for NPTL, NGPT, and the older Linuxthreads, although they were not produced by an independent institution. In the fall of 2002, Molnar and Drepper published benchmark results indicating that their own implementation is four times better than Linuxthreads and twice as good as NGPT. However, a stable NGPT has now been released and its authors are claiming better performance than NPTL. It also remains to be seen what the new Gnu Pth version will have to offer.

Thus, the race for which thread system will be used in future Linux versions is still on, at present. ■

INFO

[1] Bovet and Cesati, "Understanding the Linux-Kernel": O'Reilly, ISBN 0569-00002-2

[2] J. Cooperstein, "Linux Multithreading Advances": *http://www.oreillynet.com/pub/a/onlamp/2002/11/07/linux_threads.html*

[3] Gnu Pth: *http://www.ossp.org/pkg/lib/pth/*

[4] Portable Multithreading, Ralf S. Engelschall: *www.engelschall.com/pw/usenix/2000/pmt.pdf*

[5] Next Generation Posix Threads (NGPT): *http://www.ibm.com/developerworks/oss/pthreads*

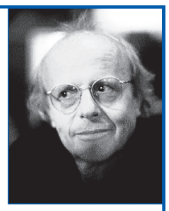
[6] Download of the current NPTL version: *http://people.redhat.com/drepper/nptl/*

[7] Drepper and Molnar, "The Native Posix Thread Library for Linux": *people.redhat.com/drepper/nptl-design.pdf*

[8] NPTL installation guide: *https://listman.redhat.com/pipermail/phil-list/2002-November/000275.html*

[9] NPTL/NGPT benchmarks by Red Hat: *http://people.redhat.com/drepper/perf-s-100000-pro.pdf*

THE AUTHOR



After studying computer science, Wolfgang Hetzler worked as a lecturer for mainframe computing. Originally being specialized in UNIX, he moved on to Linux in 1993. Wolfgang was a member of the teaching staff at the computer science department of the Technical University of Frankfurt, Germany. You can contact him at: het@het.gg.uunet.de.