Linux Threads Frequently Asked Questions (FAQ)

by Sean Walton, KB7rfa

walton@oclc.org

(Last revised 19 Sep 1996)

See also: This mirror in Italy ... may speed your access.

Caution

This FAQ is more than a two years out-of-date. POSIX-threads are now a standard part of all modern Linux distributions. The new glibc version 2 (linux libc version 6.0) is fully re-entrant and supports threads in a fully compliant manner. The default Linux thread implementation is with kernel-space threads, not user-space threads; these threads will schedule properly on an SMP architecture.

You may still find this FAQ useful if you are looking for user-space threads, DCE threads, a non-standard threads API, or for threads tools for a language other than C/C++/perl/tcl/scheme, or if you are upgrading an older system. The "LinuxThreads" package (below) has become the default package for Linux distributions, so you will probably want that if upgrading. The MIT (Provenzano) threads package is popular among some folks.

Since Sean Walton has moved on to bigger & better things, there is currently no maintainer for the FAQ. If anyone wants to bring this FAQ up-to-date, or can offer an otherwise improved and updated FAQ, please contact me, Linas Vepstas, linas@linas.org and I will post your updates and/or redirect this page to your page.

Introduction

> This FAQ is designed to answer several frequently asked questions about Linux threads. For an in-depth view of threads or Linux threads join the discussions on comp.os.linux.development.system (c.o.l.d.s).
>
> Since I am only conveying the information (and not an expert), please direct your specific questions to c.o.l.d.s or comp.programming.threads. If you have suggestions/additions/revisions to this document, please email me.

Limitations

This is **not** a discussion on the various implementations of threads: it is specifically for Linux threading. If you want to see a comparison between systems or would like to promote some other system than is intended here, write your own FAQ. Nevertheless, I welcome comments pertinent to the topic and will gladly add your input.

Credits

A special thanks to those who significantly helped me put this together: Byron A Jeff (for a lot of theory) and Steven L. Blake (for his list of sources).

INDEX

What are threads (user/kernel)?

Threads are "light weight processes" (LWPs). The idea is a process has five fundamental parts: code ("text"), data (VM), stack, file I/O, and signal tables. "Heavy-weight processes" (HWPs) have a significant amount of overhead when switching: all the tables have to be flushed from the processor for each task switch. Also, the only way to achieve shared information between HWPs is through pipes and "shared memory". If a HWP spawns a child HWP using fork(), the only part that is shared is the text.

Threads reduce overhead by sharing fundamental parts. By sharing these parts, switching happens much more frequently and efficiently. Also, sharing information is not so "difficult" anymore: everything can be shared. There are two types of threads: user-level and kernel-level.

User-Level Threads

User-level avoids the kernel and manages the tables itself. Often this is called "cooperative multitasking" where the task defines a set of routines that get "switched to" by manipulating the stack pointer. Typically each thread "gives-up" the CPU by calling an explicit switch, sending a signal or doing an operation that involves the switcher. Also, a timer signal can force switches. User threads typically can switch faster than kernel threads [however, Linux kernel threads'

switching is actually pretty close in performance].

Disadvantages. User-level threads have a problem that a single thread can monopolize the timeslice thus starving the other threads within the task. Also, it has no way of taking advantage of SMPs (Symmetric MultiProcessor systems, e.g. dual-/quad-Pentiums). Lastly, when a thread becomes I/O blocked, all other threads within the task lose the timeslice as well.

Solutions/work arounds. Some user-thread libraries have addressed these problems with several work-arounds. First timeslice monopolization can be controlled with an external monitor that uses its own clock tick. Second, some SMPs can support user-level multithreading by firing up tasks on specified CPUs then starting the threads from there [this form of SMP threading seems tenuous, at best]. Third, some libraries solve the I/O blocking problem with special wrappers over system calls, or the task can be written for nonblocking I/O.

Kernel-Level Threads

Kernel-level threads often are implemented in the kernel using several tables (each task gets a table of threads). In this case, the kernel schedules each thread within the timeslice of each process. There is a little more overhead with mode switching from user->kernel-> user and loading of larger contexts, but initial performance measures indicate a negligible increase in time.

Advantages. Since the clocktick will determine the switching times, a task is less likely to hog the timeslice from the other threads within the task. Also I/O blocking is not a problem. Lastly, if properly coded, the process automatically can take advantage of SMPs and will run incrementally faster with each added CPU.

Combination

Some implementations support both user- and kernel-level threads. This gives the advantages of each to the running task. However, since Linux's kernel-level threads nearly perform as well as user-level, the only advantage of using user-threads would be the cooperative multitasking.

What are the common problems with threads?

Several problems with threads originate from a classic view and its intrinsic concurrency complexity.

Classic View

In many other multithreaded OSs, threads are not processes merely parts of a parent task. Therefore, the question of "what happens if a thread calls fork() or (worse) if a thread execve()'s some external program" becomes problematic: the **whole task** could be replaced. The POSIX standard defines a thread calling fork() to duplicate only the calling thread in the new process; and an execve() from a thread would stop all threads of that process.

Having two different implementations and schedulers for processes is a flaw that has perpetuated from implementation to implementation. In fact, some multitasking OSs have opted **not** to support threads due to these problems (not to mention the effort needed to make the kernel and libraries 100% reentrant). For example, the POSIX Windows NT appears not to support threads [author note: it *was* my understanding that they *did* support threads... that's part of Win32NT].

Most people have a hard enough time understanding tasks, never mind "chopped up tasks" or threads. The first problem while programming is answering the question: "What *can* be threaded in my app?". That, in itself, can be very laborious (see section on "What kinds of things should be threaded/multitasked?").

Another problem is locking. All the nightmares about sharing, locking, deadlock, race conditions, etc. come vividly alive in threads. Processes don't usually have to deal with this, since most shared data is passed through pipes. Now, threads can share file handles, pipes, variables, signals, etc. Trying to test and duplicate error conditions can cause more gray hair than a wayward child.

Does Linux support threads?

Yes. As of 1.3.56, Linux has supported kernel-level multithreading. There also have been user-level thread libraries around as early as 1.0.9.

Are Linux threads the same as other implementations?

No. They are better -- while mostly keeping the same API. As stated above, most multithreaded OSs define a thread **separately** from processes. Linus Torvalds has defined that a thread is a "context of execution" (COE). This means that only one process/thread table and one scheduler is needed.

Traditionally, a thread was just a CPU (and some other minimal state) state with the process containing the remains (data, stack, I/O, signals). This would lend itself to very fast switching but would cause basic problems (e.g. what do "fork()" or "execve()" calls mean when executed by a thread?).

Consider Linux threads as a superset of this functionality: they still can switch fast and share process parts, but they can also identify what parts get shared and have no problems with execve() calls. There are four flags that determine the level of sharing:

```
#define CLONE_VM 0x00000100

#define CLONE_FS 0x00000200

#define CLONE_FILES 0x00000400

#define CLONE_SIGHAND 0x00000800

#define CLONE_PID /* not fully implemented */
```

There has been a lot of talk about "clone()". The system call (please note: ***low level***) clone() is an extension to fork(). In fact, clone(0) == fork(). But with the above #define's, any combination of the VM, filesystem, I/O and signal handlers may be shared.

Is the kernel 100% reentrant?

No. It's getting there though. As of this date 12-Sep-96, some of the drivers are still not reentrant. What does this mean? It means that threads making various system calls will block other threads until the call is completed. Will it crash? No. When will the fine-threading (not treating system calls as a single operation) work? That's currently in the works.

Do the libraries support multithreading?

No. These are more likely to crash. There are several objects that are hidden from applications and may be shared inadvertently (e.g. FILE* or emulated NDP registers). Additionally there are conflicts between some libraries. For example, SVGAlib & LinuxThreads both use SIGUSR1 and SIGUSR2 (application-reserved signals). If an app were to use these libraries together, the app would minimally have problems (or likely crash) and debugging would be a sequel to "Friday the 13th".

Several individuals are working hard to get the libraries in sync with this new functionality. The initial work is to provide wrappers around some of the critical, data-shared functions (e.g. open, close). In fact some threads libraries have work-arounds for these as well.

Please note that revising the standard libraries is *no small task* (gratuitous pun;). If you want to help, contact ----@----.----.

What kinds of things should be threaded or multitasked?

If you are a programmer and would like to take advantage of multithreading, the natural question is what parts of the program should/ should not be threaded. Here are a few rules of thumb (if you say "yes" to these, have fun!):

- Are there groups of *lengthy* operations that don't necessarily depend on other processing (like painting a window, printing a document, responding to a mouse-click, calculating a spreadsheet column, signal handling, etc.)?
- Will there be few locks on data (the amount of shared data is identifiable and "small")?
- Are you prepared to worry about locking (mutually excluding data regions from other threads), deadlocks (a condition where two COEs have locked data that other is trying to get) and race conditions (a nasty, intractable problem where data is not locked properly and gets corrupted through threaded reads & writes)?
- Could the task be broken into various "responsibilities"? E.g. Could one thread handle the signals, another handle GUI stuff, etc.?

Are there threading libraries? Where?

Yes, there are several. Here are a several URLs Steven Blake (2-Aug-1996) has compiled for our

use. Please note that I have emailed the authors (if known) to get more information about their libraries.

Linux Threads Packages

Title:	Bare-Bones Threads
Author:	Christopher Neufeld
Repositories:	[Documentation] [Source]
API:	Non-standard
Description:	This is a very basic, bare-bones threading package which has been tested on both single-CPU and SMP Linux boxes.
License:	GPL(? Source is included)

Title:	DCEthreads
Author:	Michael T. Peterson
Repositories:	[Documentation] [Source]
API:	POSIX 1003.4c Draft 4 (?)
Description:	PCthreads (tm) is a multithreading library for Linux-based Intel systems and is based on the POSIX 1003.1c standard. The kit contains the sources for the library (libpthreads), a build environment for building both ELF and A.OUT versions of the library, and a complete set of man pages for all POSIX .1c functions.
License:	GPL(? Source is included)

Title:	FSU Pthreads
Author:	Frank Mueller
Repositories:	[Documentation] [Documentation mirror] [Source] [Source mirror]
API:	POSIX 1003.1c
Description:	Pthreads is a C library which implements POSIX threads for SunOS 4.1.x, Solaris 2.x, SCO UNIX, FreeBSD and Linux. Used for GNU Ada Runtime; partial libc support.
License:	Gnu Public Library License

Title:	JKthread
Author:	Jeff Koftinoff

Repositories:	[Documentation] [Source]
API:	Non-standard
Description:	This is an experiment with the Linux 2.0 clone() call to implement usable kernel threads in a user program. These jkthreads have an API that has NOTHING to do with pthreads or Win32 threads or BeBox threads.
License:	GPL (? Source is included)

Title:	LinuxThreads
Author:	Xavier Leroy
Repositories:	[Documentation] [Source]
API:	POSIX 1003.1c
Description:	LinuxThreads is an implementation of the Posix 1003.1c thread package for Linux. Unlike other implementations of Posix threads, LinuxThreads provides kernel-level threads: threads are created with the new clone() system call and all scheduling is done in the kernel.
License:	GNU LGPL

Title:	LWP
Author:	Stephen Crane
Repositories:	[Documentation] [Source]
API:	Non-standard
Description:	A small portable lightweight process library for sun[34], mips-ultrix, 386BSD, HP-UX and **Linux**. (Man pages included)
License:	GPL

Title:	PCthreads
Author:	Michael T. Peterson
Repositories:	[Documentation] [Source]
API:	POSIX 1003.1c
Description:	User-space pthreads library; includes non-blocking select(), read(), and write(). Man pages included. Requires DCEThreads.
License:	GPL (? Source is included)

Title:	Provenzano Pthreads

Author:	Christopher A. Provenzano
Repositories:	[Documentation] [Source]
API:	POSIX 1003.1c subset (lacks thread cancellation)
Description:	User-space pthreads library distributed with Linux libc source but may not be built by default.
License:	GPL (?)

Title:	QuickThreads
Author:	David Keppel
Repositories:	[Documentation] [Source]
API:	Non-standard
Description:	A portable user-space threads package. Documentation written in PostScript.
License:	Freeware (source).

Title:	Radke Threads
Author:	Thomas Radke
Repositories:	[Documentation] [Source]
API:	Non-standard
Description:	User-level threads package included with patches to Linux kernel to support kernel threading. (Includes man pages)
License:	GPL (? Source is included)

How are Linux kernel threads accessed?

Since kernel threads are individual tasks with various shared parts, the question naturally arises: how are the threads associated with the parent and how are they accessed? There appear to be two ways: through the language or through the kernel.

Language Access

There exist several languages that support threads intrinsicly: Modula-3, Java, Python 1.4, Smalltalk/X, Objective-C/Gnustep and Ada. Each have language elements to program/access individual threads. All of these languages are available to the Linux community. However, they only support user threads; no "clone()" calls are made to the new Linux kernels. There appears to be effort, however, in revising these languages to support the newer kernels.

Kernel Access

Each PID is 32bits, wrapping (modulus) at 30000 for really old software. If CLONE_PID is **not** used, each thread will get its own PID like any other process. However, if the PID is to be shared, the kernel uses the upper 16bits to assign the thread ID (TID) [please note that this is probably **not** in the 2.0.* kernel version; we'll see it in 2.1.* for sure.]

Furthermore, each process has at least one thread (the parent). Each new thread will be assigned a TID beginning with 1. A TID of 0 (e.g. 0x0000FFFF mask) will address all threads within a process. Suppose an app has three threads (parent and two task managers) and the threads share the parent's PID. Suppose, the PIDs for each might be 0x00011234 (parent), 0x00021234 (child thread #1) and 0x00031234 (child thread #2). Each thread can be accessed or signaled individually -or- the whole task could be addressed with 0x00001234 (note that the first four digits are zero masking the TIDs).

It is the intent that the long format will work with existing apps. And, older apps that signal the whole task will still work (by accessing the whole task at once). However a shorthand has been proposed: PID.TID (e.g. 46.2 would be the second thread of PID 46).

Is there a system call I can use to access kernel threads?

Recent versions of glibc has clone() defined thus:

```
int clone(int (*fn)(), void **stack, int flags, int argc,... /* args */);
```

fn	The thread routine
stack	The thread's stack
flags	Flags as defined above
argc	number of remaining parameters
/* args */	the parameters expected by the thread.

Are there ways currently to determine thread schedule ordering?

Not 100%. There are realtime extensions in the 2.0.0 kernels that will grant finer-tuned control over threads/tasks via `sched_setscheduler'. [Author's note: I have no idea about this. I will update this as I get more info.]

Are there languages that support threads?

Currently, Ada, Modula-3, Python 1.4, SmallTalk/X, Objective-C/ Gnustep and Java have intrinsic language elements to support threads (for example, Ada has the 'Select' statement which is a "thread guard"). However, the Linux implementations of these languages do not yet support

the new kernel threads.

Naturally, C and C++ can make calls to spawn threads and processes, but there are no language elements to support them beyond these system calls. See below for a listing of available languages:

Threading Languages

Titles:	Ada/Ed
Author:	New York University
Repositories:	[Documentation] [Source]
Newsgroup:	comp.lang.ada
Threads Lib:	LinuxThreads
Description:	Ada/Ed is a translator-interpreter for Ada. It is intended as a teaching tool and does not have the capacity, performance or robustness of commercial Ada compilers. Ada/Ed was developed as a long-range project in language definition and software prototyping.
License:	GPL

Title:	Gnat
Author:	New York University
Repositories:	[Documentation] [Source][SVGA Bindings]
Newsgroup:	comp.lang.ada
Threads Lib:	LinuxThreads
Description:	GNAT is the Ada 95 compiler produced by a collaboration between a team at New York University and the Free Software Foundation, 'owner' of the GNU software project.
License:	GPL

Title:	Guavac & Kaffe (Java)
Author:	???
Repositories:	[Guavac Documentation] [Kaffe Documentation] [Gauvac Source] [Kaffe Source]
Newsgroup:	
Threads Lib:	???

Description:	Guavac is a new compiler for the Java language, written by Effective Edge Technologies and distributed under the Gnu Public License. You should feel free to use, copy and modify it, based on the terms in the COPYING file included in this distribution. Kaffe is a virtual machine design to execute Java bytecode. Unlike other virtual machines available, this machine performs "just-in-time" code conversion from the abstract code to the host machine's native code. This will ultimately allow execution of Java code at the same speed as standard compiled code but while maintaining the advantages and flexibility of code independence.
License:	GPL

Title:	Modula-3/m3gdb
Author:	DEC Systems Research Center
Repositories:	[Documentation] [Source]
Newsgroup:	comp.lang.modula-3
Threads Lib:	(Uses own?)
Description:	Compiler, tools, applications and libraries for Modula-3, a simple and efficient modular, imperative language. Modula-3 has objects, threads, exceptions and generics. The libraries include X toolkits, a user interface builder, an embedded interpreted language and network objects. m3gdb is a GPL debugger for Modula-3.
License:	(see copyright, freely usable and redistributable)

Title:	Objective-C/Gnustep
Author:	www.gnustep.org
Repositories:	[Documentation] [Source]
Newsgroup:	comp.lang.objective-c
Threads Lib:	pthreads (user-level [kernel-level in development])
Description:	objc-shared-patches contains a complete source of GNU Objective-C runtime and a diff file for libobjects-0.1.14, both patched to generate a shared Linux ELF library.
License:	GPL

Title:	Python 1.4
Author:	www.python.org
Repositories:	[Documentation][Binary][Source]

Newsgroup:	comp.lang.python
Threads Lib:	POSIX threads (Python Makefile supports user-/ kernel-level)
Description:	Python is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java.
License:	Freeware

Title:	Sather
Author:	ICSI
Repositories:	[Documentation] [Source]
Newsgroup:	comp.lang.sather
Threads Lib:	POSIX Threads
Description:	Sather is an object oriented language designed to be simple, efficient, safe, flexible and non-proprietary. One way of placing it in the "space of languages" is to say that it aims to be as efficient as C, C++, or Fortran, as elegant as and safer than Eiffel, and support higher-order functions and iteration abstraction as well as Common Lisp, CLU or Scheme.
License:	Freeware (?)

Title:	SmallTalkX
Author:	???
Repositories:	[Documentation] [Source]
Newsgroup:	comp.lang.smalltalk
Threads Lib:	(Internal implementation of threads.)
Description:	SmallTalk interpreter for X11. SmallTalk is an object-oriented, interpreted programming language. Often it is used in simulations or rapid prototyping.
License:	Noncommercial (see `LICENSE')

How does one debug threads?

(Author's note: Good question. Any thoughts?)

Modula-3

Modula-3 supports user-level thread breakpoints. For further help here, please refer to the Modula-3 specifications.

C/C++ (and anything compatible with gdb)

Gdb supports children processes and threads equally (since they are based on the task paradigm) if they do not share PIDs. Those threads which share PIDs can be accessed using the good-old-fashioned printf debugging (for now). (Can someone guide me on accessing specific processes within gdb?)

What do the individual flags mean and do in clone()?

The clone() system call has several flags that will indicate how much will be shared between threads. Below you will find a table listing each flag, its function & its implementation status.

Flag	Status	Description
CLONE_VM	Done	Share data and stack
CLONE_FS	Done	Share filesystem info
CLONE_FILES	Done	Share open files
CLONE_SIGHAND	Done	Share signals
CLONE_PID	Almost Done	Share PID with parent (problems with /proc and signals go to parent)

What applications or libraries currently use threads?

There are few applications and libraries that currently use threads (user or kernel) which have been ported to Linux. However, upon becoming known, they will appear in this list. This list will also hopefully indicate which apps need which threading library.

(Author's Note: If you know of any applications or libraries that should appear in this list, please send me information about it in the same format.)

Thread-Using Packages/Libraries

Library:	Adaptive Communication Environment (ACE)
Author:	Douglas C. Schmidt
Repositories:	[Documentation] [Source] [Mirror]
Threads Lib:	LinuxThreads
	The ADAPTIVE Communication Environment (ACE) is an object-oriented programming toolkit for concurrent network applications and services. ACE encapsulates user-level UNIX and Win32 (Windows NT and Windows '95) OS

Description:	mechanisms via portable, type-secure, efficient, and object-oriented interfaces. In addition, ACE contains a number of higher-level class categories and network programming frameworks. The following diagram illustrates the key components in ACE and their hierarchical relationships.
License:	[License]

Title:	AolServer
Author:	www.aolserver.com
Repositories:	[Source]
Threads Lib:	(Uses internal system calls to clone()?)
Description:	A free webserver (multiplatform) [Feature Sheet]
License:	[License]

Title:	Executor
Author:	www.ardi.com
Repositories:	[Source]
Threads Lib:	???
Description:	A100% native software Macintosh emulator for PCs. Executor lets you read and write Mac-formatted high-density floppies and Mac SCSI drives, read Mac CDs, and run many Macintosh programs.
License:	(commerial software)

Where can I learn more about threads?

Here is a list of URLs, newgroups, etc. where you can learn more about thread programming:

Newsgroup: comp.os.linux.development.system

Newsgroup: comp.programming.threads

Sun: Workshop Developer Products--Threads

IEEE Parallel & Distributed Technology

---

web page titled "GNU Free Documentation License".