UCRL-WEB-201477

Pthreads Overview (for LC)

Table of Contents

Preface

Scope: This Pthreads Overview (for LC) provides a coordinated, conceptual introduction to using POSIX threads (pthreads) in parallel programs, compares their benefits and their known dangers, and spells out LC-relevant implementation details (such as local threads-related environment variables, locally applicable thread synchronization techniques, and portability-inhibiting differences in the way LC vendors have developed pthreads support). This overview is intended to summarize otherwise scattered backgound information on pthreads, and to complement the specific programming examples available in LC's Pthreads Tutorial (URL: http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html) by Blaise Barney.

Availability: When the programs or techniques described here are limited by machine, those limits are included in their explanation. Otherwise, they run under any LC UNIX system.

Consultant: For help contact the LC customer service and support hotline at 925-422-4531 (open e-mail: lc-hotline@llnl.gov, SCF e-mail: lc-hotline@pop.llnl.gov).

Printing: The print file for this document can be found at:

```
OCF: http://www.llnl.gov/LCdocs/pthreads/pthreads.pdf
SCF: https://lc.llnl.gov/LCdocs/pthreads/pthreads_scf.pdf
```

Introduction

Besides allowing the execution of multiple processes at the same time, UNIX also supports multiple independent flows of control (or "threads") within a single process (sharing the process resources). Using such POSIX-compliant threads (or "pthreads") is a much more fine-grained approach to computing several tasks at once, but often a more intricate one as well. This overview provides a systematic conceptual framework for using POSIX threads on LC machines.

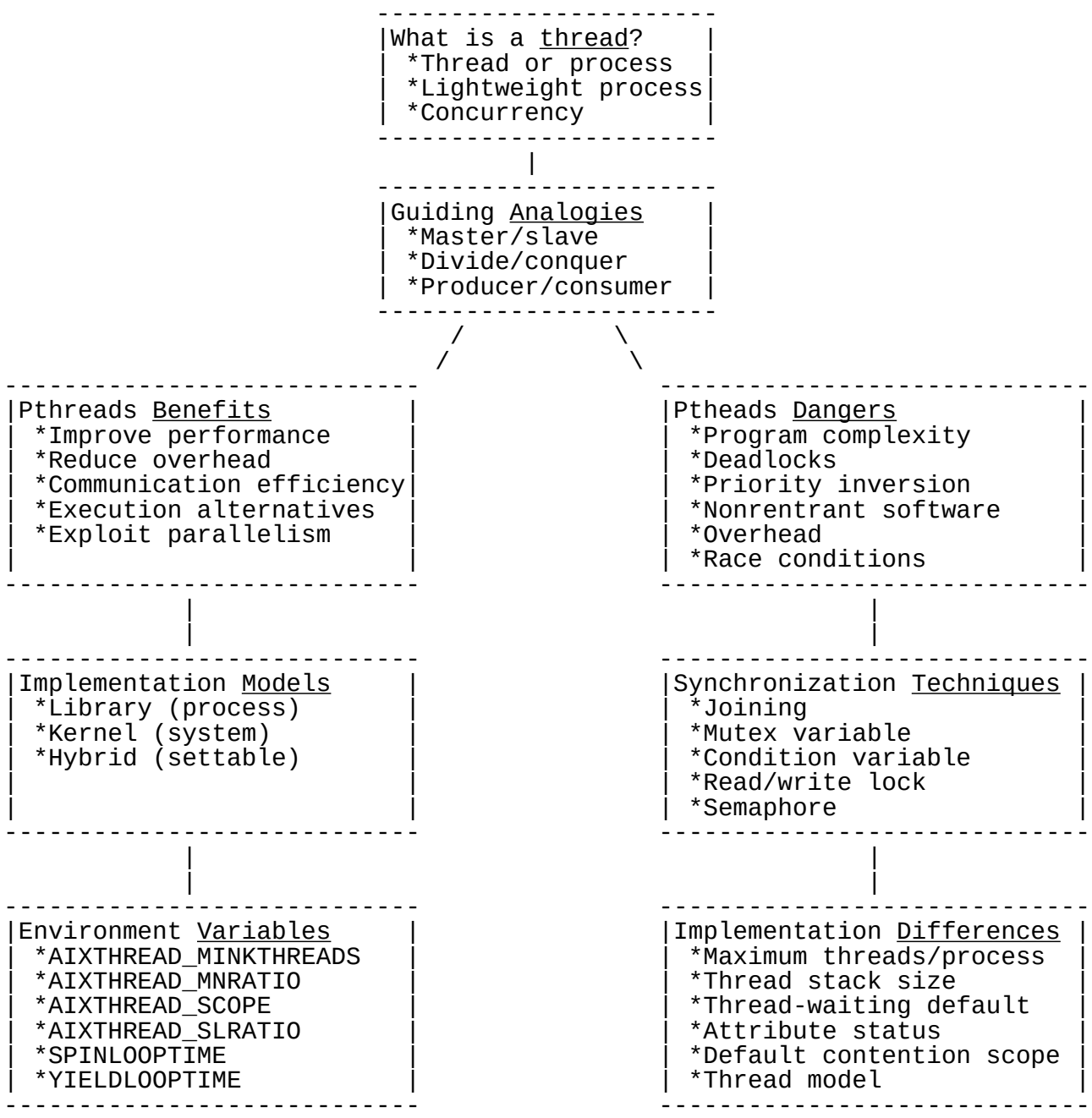
This manual begins with a comparison of processes and threads, of C and Fortran support for thread use, and of concurrency and parallelism as program-design issues. Three alternative analogies (e.g., master/slave) for dividing tasks among the threads of a multithreaded program are presented and compared. And the potential benefits of using pthreads are cataloged.

There are three different ways to implement threads, and another section compares the features of these "thread implementation models." As is typical of UNIX systems, environment variables are an important part of any actual threads implementation, so one section explains their role in tuning threads behavior under AIX on LC's IBM machines.

Threads have their problems as well as their benefits. A later section explains the most common pitfalls of multithreaded programs, a survey that serves as good background for a detailed comparison of five ways to synchronize thread processing to avoid or minimize those pitfalls in practical programs. The strengths and weaknesses of each thread synchronization technique are spelled out, followed by links to specific cases and coding examples already published elsewhere.

This overview ends with a summary of locally relevant differences among the pthreads implementations provided by different vendors on different LC machines. Because of the growing importance of Linux (CHAOS, and its SLURM resource manager) at LC, cross reference links to additional details about local Linux pthreads implementations are featured here. CHAOS 2.0 replaces the former LinuxThreads pthreads library with the more sophisticated Native POSIX Threading Library, so this section also compares their relevant features. Also included is an introduction to Sphinx, an LLNL microbenchmark suite for performance testing of pthreads.

The diagram below provides both a quick visual survey of the topics discussed in this guide to pthreads and a comparative analysis of the major issues, positive and negative, that confront everyone who tries to use pthreads effectively. Each block in the diagram links directly to the corresponding section later in the text where specifics are explained.


What is a thread?
*Thread or process
*Lightweight process
*Concurrency
Guiding Analogies
*Master/slave
*Divide/conquer
*Producer/consumer
Pthreads Benefits
*Improve performance
*Reduce overhead
*Communication efficiency
*Execution alternatives
*Exploit parallelism
Ptheads Dangers
*Program complexity
*Deadlocks
*Priority inversion
*Nonrentrant software
*Overhead
*Race conditions
Implementation Models
*Library (process)
*Kernel (system)
*Hybrid (settable)
Synchronization Techniques
*Joining
*Mutex variable
*Condition variable
*Read/write lock
*Semaphore
Environment Variables
*AIXTHREAD_MINKTHREADS
*AIXTHREAD_MNRATIO
*AIXTHREAD_SCOPE
*AIXTHREAD_SLRATIO
*SPINLOOPTIME
*YIELDLOOPTIME
Implementation Differences
*Maximum threads/process
*Thread stack size
*Thread-waiting default
*Attribute status
*Default contention scope
*Thread model

What is a Thread?

DEFINITIONS.

Every discussion of POSIX threads begins with a comparison between a process and a thread:

A PROCESS is created by the operating system as a set of physical and logical resources to run a program. A every process includes:

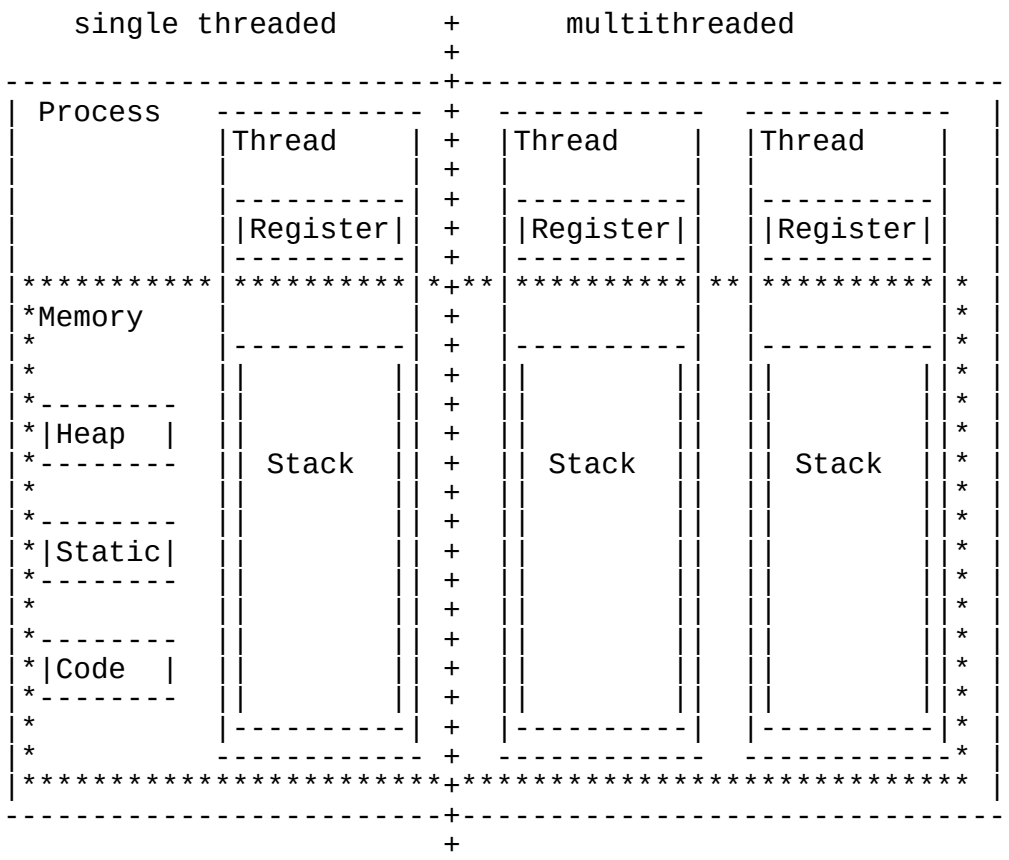
- heap, static, and code memory.
- registers to manage code execution.
- a stack.
- environment information, including a working directory and file descriptors.
- process, group, and user IDs.
- interprocess communication tools and shared libraries.

A THREAD is the execution state of a program instance, sometimes called an independent flow of control. It has just enough properties to allow it to run independently:

- registers to manage code execution.
- a stack.
- scheduling properties (such as priority).
- its own set of signals.
- some thread-specific data.

Hence a thread is sometimes referred to as a "lightweight process" (LWP), although on IBM systems a lightweight process is usually taken to mean something more specialized (a "kernel thread," as discussed later (page 12)).

This diagram shows the process/thread relationship:



A traditional process had a single thread (left side). A multithreaded process (right side) shares its resources among several threads, all executing in the same address space but each with its own program counter and point of execution. Because all threads within a process share its resources:

- Changes made by any one thread (such as closing a file) will affect all other sibling threads.
- Two pointers with the same value in different threads point to the same data.
- Different threads can read from and write to the same memory locations, often convenient but always a danger to be managed carefully by the threads programmer.

The process/thread relationship is not purely academic. In many situations it has important practical consequences that affect:

- SCHEDULING of threads.
  How threads compete for scheduling by the operating system can depend on which process they belong to, and users can influence this dependence by setting various environment variables (as several later sections explain).

- SYNCHRONIZATION of threads.
  Coordinating access by many threads to the same (within-process) memory locations and orchestrating how threads exchange information among themselves while they run is so important that the pthreads library provides three separate sets of thread-synchronization primitives, on which several more synchronization methods are built. See the "Synchronization" section below (page 20) for a comparison.

Another, more subtle example of how a process interacts with its threads in practical ways involves signal handling. Under UNIX, when a process receives a signal the kernel stops the process, executes a designated piece of code called a signal handler, and as soon as the handler completes it resumes the process just where it stopped before the signal arrived. Handlers are assigned to specific signals through a signal-handler table, and the table applies to *all* threads in the process. Any thread can change the handler assigned to a signal, but such a change affects all other threads too and is overwritten by the next thread that also tries a handler reassignment. On the other hand, each separate thread maintains its own *signal mask*, another table that specifies which signals it will receive and which it will ignore. So signal selection is per thread, while signal handling is per process.

LIBRARIES.

Unfortunately, threads implementations often vary from one hardware vendor to another (for a comparison of the versions found on LC machines, see a later section (page 27) of this manual). Theoretically, a portable, vendor-neutral solution to this problem has been provided by the IEEE POSIX 1003.1c standard for POSIX threads or pthreads. IBM's AIX system 4.2 implements draft 7 of this standard, while AIX 4.3 implements draft 10 (the version that is included in the UNIX 98 standard). For the C language, this standard application programming interface for threads appears as a header (include) file called pthreads.h and a linkable library called libpthreads.a. For Fortran, XL Fortran V5 provides an interface to the portable thread library called f_pthread, implemented as a Fortran 90 module. Prefixing f_ before the corresponding C library routine name or type definition provides the Fortran counterpart (for example, pthread_create() becomes F_PTHREAD_CREATE()). Using xlf90_r or xlf_r invokes the thread-safe library for Fortran. Pthreads are compatible with the local gang scheduler; see the "Thread Use" section of LC's Gang Scheduler User Guide (URL: http://www.llnl.gov/LCdocs/gang) for details. See the 1999 LC Pthreads Tutorial (URL: http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html) for explicit coding examples using routines from the pthreads library.

CONCURRENCY.

Whenever a process can have multiple threads, two related concepts are often confused:

CONCURRENCY

> Tasks are concurrent when they can run independently in any order, including at the same time. In a multithreaded process, each thread can reach the CPU and execute its code with no predefined order (which can sometimes give rise to problems (page 18) requiring programmers to install turn-taking features in the software).

PARALLELISM

> Tasks run in parallel when there are multiple CPUs available so that several tasks can run at the same time. This means that while not all concurrent programs are parallel, all parallel programs are concurrent. LC's production machines are all "symmetric

multiprocessors," which means that essentially all of the processors are identical and can run the same tasks. This allows great flexibility in running multithreaded processes with a high degree of parallelism (many threads executing at once).

Analogies Guiding Thread Use

Three standard analogies suggest three different ways to use multiple threads when you are designing a multithreaded program.

Master/Slave (Boss/Worker)

A single master thread receives input or requests and then creates multiple slave threads and assigns the work to them. The slave threads run independently of each other (concurrently, and perhaps in parallel), but the master typically controls how many slaves there are and what each slave does. The number of slaves may be fixed or variable.

A familiar computing example of this case is the print-job spooler. As the master, the spooler receives print requests, chooses a printer, and assigns to a slave thread the handling of flow control and other printing details. The master may even cancel slave threads or reassign their jobs.

Divide/Conquer (Work Crew)

Many individual threads process related work independently (and, on an SMP system, in parallel), but there is no master thread coordinating them. Again, the number of cooperating peer threads may be fixed or variable.

A simple computing example of this case is a multitheaded version of the UNIX grep command. Here many threads each take a file from a specified pool, search it for a specified pattern, send the results to a common output device, and then return to the pool to perform another search until the pool is empty. No thread is in charge of the others.

Producer/Consumer (Pipelining)

An item is passed from one thread to the next in a series. Each thread performs one stage in a production-line process, then passes the item along to have the next stage performed. The treads run independently although they get work from each other.

A familiar computing example of this case involves connecting several UNIX commands with "pipes." Each independent utility processes the output (perhaps a list of files) from the command before it and then passes it on to the next in the series. In a parallel environment, these steps (reading, changing, writing) could all be underway at the same time.

Benefits of Pthreads

Using POSIX threads (instead of simply using many single-threaded traditional processes) offers several potential benefits and poses several likely dangers. This section summarizes the benefits, while a later section (page 18) summarizes the dangers and leads into a discussion of how to handle them.

Among the reasons to use pthreads are:

IMPROVE PROGRAM PERFORMANCE.
Using multiple threads allows a program to improve performance in several ways that take advantage of independent thread execution. For example,
(a) some threads can perform long or interrupted I/O system calls while others focus on CPU-intensive work separately,
(b) some threads can handle high-priority tasks while other look after less important background work, and
(c) handling asynchronous events can be spread among theads so that the work is interleaved and more flexible.

REDUCE SYSTEM OVERHEAD.
Creating multiple threads within a single process requires less processing and uses less memory than creating a corresponding number of separate whole processes. Significant timing differences between calls to the fork() subroutine and calls to the pthreads_create() subroutine are one indication of how system resources are conserved by using threads.

IMPROVE COMMUNICATION EFFICIENCY.
Interthread (intraprocess) communication is often more efficient and easier to carry out than interprocess communication. Unfortunately, in practice, this comparison depends heavily on hardware details. The actual bandwidth available for intraprocess and for interprocess communication varies by vendor and even by model of chip for the same vendor, so the relative efficiencies can change between platforms and with every upgrade to a given platform.

INCREASE EXECUTION ALTERNATIVES.
Multithreaded programs will run successfully on even on a single processor, and, without recompiling, they will run more flexibly on a multiprocessor system.

EXPLOIT POTENTIAL PARALLELISM.
On SMP systems, such as the production machines at LC, running many threads simultaneously (parallelism) is the big potential benefit from having many independent threads available (concurrency). Sometimes even simple program designs can benefit greatly from the high CPU utilization possible in a massively parallel environment.

Alternative Thread Implementation Models

The "contention scope" of a thread is the set of other threads with which it competes for scheduling, for CPU time. A thread's contention scope is determined by the "thread model" that the current operating system implements. Three different thread models are possible, and IBM's AIX operating system has implemented all of them in one of its versions or another.

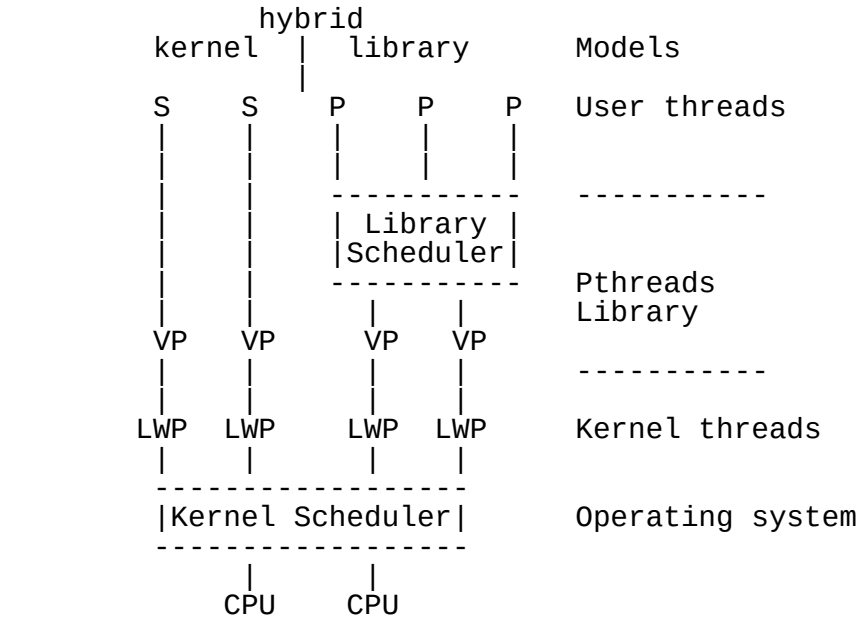
This table briefly summarizes the alternative thread models and contention scopes. The text below the figure then explains and compares all of them in detail:

Thread Model	Contention Scope
M:1 (library model)	process (local)
1:1 (kernel model)	system (global)
M:N (hybrid model)	either (settable)

Every operating system recognizes a set of entities that it schedules. These kernel-scheduled entities are called lightweight processes (LWPs) or sometimes (confusingly) "kernel threads." Conceptually distinct from these are the user threads (the data structures in process address space) that programmers plan and their programs deploy. User threads must map onto LWPs (kernel threads) to execute.

The operating system's thread model specifies just how user threads map onto kernel threads, and hence just how they compete with each other to run. The mapping of user threads to kernel threads occurs by means of a pthreads-library implicit entity called a "virtual processor" (VP), which acts for a user thread the way that a real CPU acts for a kernel thread.

This diagram helps clarify the several possible ways that these entities can interact, all specified by the current thread model (details for each possible model follow):



M:1 (Library) Model

In the M:1 or library model, each process has a kernel thread. All user threads are mapped to the single kernel thread that belongs to their process. Hence all user threads take turns running on just one virtual processor (VP), as shown on the right side of the figure above. The mapping is handled entirely by a library scheduler. Library-scheduled user threads compete only with other threads from the same process for CPU time: they have a "process contention scope." Switching a kernel thread (LWP) from one user thread to another at various times during execution is called "context switching," and is essential for implementing any M:1 thread model (because there are not enough LWPs to go around). Because it requires context switching, the library model has obvious inefficiencies, but it can be used on any system including a single-CPU system.

1:1 (Kernel) Model

In the 1:1 or kernel model, each user thread has its own kernel thread. Because the mapping is one-to-one, each user thread also has its own virtual processor, as shown on the left side of the figure above. In this model, user thread "programming facilities" are handled directly by the kernel threads to which they map, not by the threads library. User threads here are scheduled directly by the operating system, and such kernel-scheduled threads therefore compete with all other threads on the system, not just intraprocess threads, for CPU time: they have a "system contention scope." Because user threads and kernel threads are evenly matched, there is no need for the overhead costs of context switching under this model. But it clearly limits the number of simultaneous user threads the system can accept.

M:N (Hybrid) Model

In the M:N or hybrid model, M user threads are mapped onto a pool of N kernel threads serviced by N virtual processors. A user thread might be bound to a specific kernel thread and VP, but more often it is multiplexed over a set of shared kernel threads and VPs (so that both halves of the figure above are in play at once). The mapping is handled partly by a library scheduler. Depending on how the threads are mapped, they may compete systemwide (system scope) or they may compete only against others threads within the same process (process scope). In general, switching a kernel thread (LWP) from one user thread to another at various times during execution (context switching) is essential for implementing an M:N thread model as it was for the M:1 model, and for the same reason. Because of the flexibility of the context switching involved, however, the M:N model is usually regarded as the most efficient thread model. Under IBM's AIX system, the default M:N ratio is 8:1 and the default contention scope is process (local) scope, but you can change either default by setting the appropriate pthreads-relevant environment variables (page 15) (see the next section).

Environment Variables for Threads

The AIX 4.3.1 M:N pthreads implementation model provides several environment variables with which you can tune the performance of your pthreads application program on IBM SP systems. These environment variables can be set by end users (some have defaults) and are examined whenever a process initializes. So the best usage strategy is to develop a front-end script in which you set these environment variables (if needed) before you invoke your binary executable.

An example of using these pthreads-relevant environment variables occurs if you migrate an application code from another vendor's machine to the IBM AIX environment. The ratio of pthreads (user threads) to kernel threads may differ on the two platforms. If this changed ratio degrades your code's performance, you can alter the way threads are created within your code or instead try setting the AIXTHREAD_SCOPE environment variable to S (for system contention scope, see below) or try setting the AIXTHREAD_MNRATIO environment variable to a ratio other than 8:1 (again, details below).

The six AIX pthreads-relevant environment variables are:

AIXTHREAD_MINKTHREADS

> overrides the AIXTHREAD_MNRATIO environment variable (next). This allows you to manually specify the minimum number of active kernel threads (default follows from MNRATIO). The library scheduler will not reclaim kernel threads below this number.

AIXTHREAD_MNRATIO

> specifies the ratio of pthreads (M) to kernel threads (N). AIXTHREAD_MNRATIO is examined when the system creates a pthread to determine if a kernel thread should also be created to maintain the correct ratio. You can set this environment variable by supplying a value of the form
>
> ```
> p:k
> ```
>
> where k is the number of kernel threads the system uses to handle p (user) pthreads. You may specify any positive integer for p and k, but these values are used in a formula that employs integer arithmetic and this results in the loss of some precision when big numbers are specified. (See also AIXTHREAD_MINKTHREADS, above.)
>
> Defaults:
> If k is greater than p, the ratio is treated as 1:1.
> If you specify no value, the default depends on the default contention scope.
> If system scope contention is the default, the ratio is 1:1.
> If process scope contention is the default, the ratio in 8:1.

AIXTHREAD_SCOPE

> sets the contention scope of pthreads created using the default pthread attribute object (for background on contention scope, see the Alternative Thread Implementation

Models (page 12) section preceding this one). You can specify either of two exclusive values for this variable:

P indicates process scope (the default).

S indicates system scope.

AIXTHREAD_SLRATIO

determines the number of kernel threads used to support local pthreads sleeping in the library code while awaiting a pthread event, for example, attempting to obtain a mutex (discussed in the Synchronization (page 20) section below). The reason to maintain kernel threads for sleeping pthreads is that, when the awaited pthread event occurs, the pthread will immediately need a kernel thread to run on. Using a kernel thread that is already available is more efficient than creating a new kernel thread after the event has taken place.

You can set this environment variable by supplying a value of the form

```
k:p
```

where k is the number of kernel threads to reserve for every p sleeping (user) pthreads. WARNING: the relative positions of k and p are reversed here from the ratio used to assign a value to AIXTHREAD_MNRATIO. You may specify any positive integer for p and k, but these values are used in a formula that employs integer arithmetic and this results in the loss of some precision when big numbers are specified. (See also AIXTHREAD_MINKTHREADS, above.)

Defaults:
If k is greater than p, the ratio is treated as 1:1.
If you specify no value, the default ratio is 1:12.

SPINLOOPTIME

(no default) specifies the number of times that the system will try to get a busy lock without taking a secondary action, such as calling the kernel to yield the processor. Manipulating SPINLOOPTIME can be helpful on SMP systems, where the lock might be held by another actively running pthread and will soon be released. On uniprocessor systems this value is ignored.

YIELDLOOPTIME

(no default) specifies the number of times that the system yields the processor when trying to acquire a busy mutex or spin lock (see the Synchronization (page 20) section below for details) before going to sleep on the lock. YIELDLOOPTIME can be helpful for complex applications where multiple locks are in use.

Under LC's CHAOS version of the Linux operating system, you can set threads-relevant environment variables directly with SRUN options when you launch your pthreads application with SRUN, the SLURM

job-control user utility. The two CHAOS/SLURM environment variables most relevant to multithreaded jobs are:

SLURM_CPUS_PER_TASK

(default is 1) allows you to assign multiple CPUs to each (multithreaded) process in your job to improve performance. SRUN's -c (lowercase) option sets this variable. See the SRUN sections of the SLURM Reference Manual (URL: http://www.llnl.gov/LCdocs/slurm) for usage details.

SLURM_OVERCOMMIT

(default is NO) allows you to assign more than one process per CPU (the opposite of the previous variable). SRUN's -O (uppercase) option sets this variable, which is *not* intended to facilitate pthreads applications. See the SRUN sections of the SLURM Reference Manual (URL: http://www.llnl.gov/LCdocs/slurm) for usage details.

Dangers of Pthreads

Using POSIX threads (instead of simply using many single-threaded traditional processes) offers several potential benefits but poses several likely dangers too. This section summarizes the dangers, while a previous section (page 11) summarizes the benefits.

Among the known problems posed by pthreads are:

PROGRAM COMPLEXITY.

(a) Intrinsic complexity:

The increased complexity of the software is the most obvious and significant problem posed by any multithreaded coding project. Using threads can sometimes simplify certain aspects of program design, but a fairly high level of expertise is required to successfully manage the interplay of a program's many threads. Several of the specific problems below result from inexpertly managing thread interaction because it is so complex.

(b) Extrinsic complexity:

Pthreads implementations also vary by vendor in subtle, troublesome ways. The availability of attributes for mutex and condition variables (see next section), for example, is unfortunately vendor dependent. So developing portable pthreads programs that can move from one LC platform to another calls for attending to and working around these vendor differences, another source of coding complexity.

DEADLOCKS.

A common side effect of the complexity of pthreads programs is deadlock. Programming errors can cause two (or more) threads to wait for each other to release a lock forever, blocking each other from executing indefinitely. Care is especially required when designing mutual-exclusion locks to handle thread synchronization, as discussed in the next section (page 20).

PRIORITY INVERSION.

Among every thread's scheduling parameters is its priority, its relative importance to run compared with competing threads. Priority inversion occurs when high-priority threads do not execute because they have unintended dependencies on one or more other, low-priority threads. Again, this is often a side effect of program complexity.

NONREENTRANT SOFTWARE.

A threaded program must use only reentrant libraries and functions, in other words, those that:

(a) do not hold static data over successive calls, and

(b) do not return a pointer to static data, and

(c) use local variables dynamically allocated from the thread's own stack, and

(d) provide a locking mechanism when working with global or shared data.

OVERHEAD.

Threads are much "lighter" than processes in operating system overhead, an often-cited benefit of the pthreads approach to parallel programming. But careless or unnecessary thread handling can still waste much processing time, especially in these three areas:

(a) Creating/terminating Threads.

Even with their relative economy over processes, threads still incur some overhead when you create and later terminate them. If you do this often, for many threads, the result can become very expensive and

degrade program performance. You should budget your thread creation and termination when you plan your appplication. For example, you could create a pool of threads once and reuse them several times to cut overhead.

(b) Context Switching.

Normally a multithreaded program has more threads than there are available CPUs, which can cause frequent thread context switching (see the Models (page 12) section above for background). For example, every time a thread blocks waiting for a lock or an I/O operation, it generates a voluntary context switch (letting another thread swap to its virtual processor). Although thread context switches are relatively lightweight, they still consume CPU time and often can be avoided.

(c) Data Sharing.

Multithreaded applications typically use locks of several kinds (compared in the Synchronization (page 20) section below) to safely share data and wait for other threads to complete. But such locks degrade overall application performance (i) a little when the locking and unlocking takes place, and (ii) a lot more when threads wait for other threads to remove a previously placed lock.

RACE CONDITIONS.

If several threads alternately put and get data from the same variable (perhaps to pass it from one to another along the stages of a software pipeline (page 10)) then in an improperly designed program success may depend implicitly on the order in which the reading and writing occurs. If the threads race to see who acts next and the "wrong" thread wins (a "race condition"), then the program may fail to behave as intended or even worse may work its way into a deadlock. Avoiding a race condition calls for somehow synchronizing the threads. This is such a common and important problem that at least half a dozen (often related) techniques have been developed to address it, and the next section (page 20) explains and compares them.

Thread Synchronization Techniques

Because race conditions and deadlocks among independent threads (see previous section (page 18)) can stop a multithreaded program or cause it to intermitantly produce faulty results, you need to overtly *synchronize* (coordinate in time) the data accesses of the threads that you deploy. There are many ways to synchronize threads. The pthreads library provides three different synchronization primitives, and at least two other effective synchronization techniques are built by further combining those primitives. The subsections of this section compare these thread synchronization techniques (8.1) and then describe the strengths and weaknesses of each one (8.2).

Comparative Chart

This chart lists five alternative ways to synchronize threads, introducing their names and comparing their basic features and roles. For more details on any technique, see its subsection below.

Technique Name	**Intended Role**	**Control Scheme**	**Activity Style**	**Best For**	**Shared Variable?**
PRIMITIVES:					
joining	wait until specified thread completes	voluntary wait	active (signal sent to waiting thread)	waiting for task completion	no
mutex variable	wait until binary variable unlocked	exclusion based access control	passive (polling needed)	synch access to shared data	yes
condition variable	wait until specified condition met	value based access control	active (signal sent to waiting thread)	waiting for task completion	yes
COMPOSITES:					
read/write lock	simultaneous reads but turn-taking writes	cond var is a read counter	active (signal sent to waiting thread)	synch access to shared data	yes (combination)
semaphore	same as joining	cond var is a task counter	active (signal sent to waiting thread)	waiting for task completion	yes (combination)

Synchronization Alternatives

Joining

SUMMARY.

Joining synchronizes the tasks that two threads perform by letting one thread, the requester, wait for the completion of the other (the target, specified by its thread ID) before continuing with its own work. The requester thread invokes subroutine pthread_join(threadid,status), which blocks the calling thread until the thread with the specified ID terminates.

STRENGTHS.

Joining threads is easy because pthread_join is a primitive included in the standard pthreads library. Joining also easily combines with other pthreads synchronization primitives (next two subsections). For example, a parent thread can wait for the completion of all of its children by joining each of them, while they invoke other techniques (below) to synchronize data access among themselves. Finally, joining is an "active" technique because the waiting thread automatically receives a signal to unblock as soon as the target thread exits.

WEAKNESSES.

(1) Threads are created in either of two states. "Detached" threads are *not* joinable; other threads cannot wait for them to end. Only "undetached" threads are joinable, allowing others to wait for them. IBM's AIX 4.3 creates all threads an undetached (joinable) by default, but this varies from one operating system and version to another (thus AIX 4.2 created all threads as detached (unjoinable) by default). You may need to specifically create threads as undetached (by using pthread_attr_setdetachstate to set the attr attribute before you invoke the pthread_create routine) if you plan to join them.

(2) Also, every undetached thread keeps its storage until the end of the whole process to which it belongs, unless another thread actually does join it. Detached threads, on the other hand, always automatically return their storage to the process heap as soon as they exit. Because having many undetached threads can be very resource expensive, some strategists urge using semaphores (page 25) (see below) to synchronize the tasks of several threads instead of joining them.

(3) In some implementations of pthreads, attempting to join a thread that happens to have ended already will cause error conditions or unpredictable behavior.

EXAMPLES.

Excellent examples of joining threads, ranging from simple waiting to complex combinations that also use locks, appear in LC's Pthreads Tutorial (URL: http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html) by Blaise Barney.

Mutex Variables

SUMMARY.

A "mutual exclusion variable" or mutex uses exclusion to force competing threads to take turns using the same data or resource. One thread creates and initializes a *binary* variable that serves as a lock. The first thread that tries to lock the mutex variable succeeds, then runs a "critical section" of code (a sequence flanked by the lock and unlock steps) to use shared storage (or to perform other work threatened by a race condition (page 18)). All other threads that try to lock the already locked mutex variable fail, and must do something else until the one successful thread completes its work and unlocks the mutex, releasing the shared resource.

STRENGTHS.

Using mutex variables to lock shared resources and protect critical sections of a program is fairly easy because the pthreads library provides primitives to create, destroy, lock, and unlock the variable used for synchronization. Another primitive, pthhread_mutex_trylock, even lets a thread check a mutex, detect its locked state, and return immediately with a "busy" error code, so that the thread can perform other work instead of just waiting on the lock (often a cause of "priority inversion (page 18)"). Combining mutex calls with other synchronization techniques is also easy and common, as shown in the examples cross referenced below.

WEAKNESSES.

(1) When used alone, a mutex is a passive synchronization device, not an active signal. It requires every thread that does not "own the lock" to enter a loop of repeated attempts to lock the mutex ("polling") until the owner unnlocks it and surrenders control to the next lucky thread. This situation, called a "spin lock," often runs fairly well on a machine with many CPUs, but it can clearly burn up many cycles if many threads are involved.

(2) A mutex variable has only two states (locked, unlocked), so only by combining it with other techniques (described in the next subsections) can it function where a counter of multiple states is needed.

(3) Mutex use is prone to implementation-dependent problems. For example, some pthreads implementations (but fortunately not AIX) interpret a second attempt by a thread (or even its children) to relock a mutex that they have already locked as a deadlock situation. The more fine-grained your locking approach (such as locking every item in a database), the more mutexes are in play, and the more likely you are to hit an unintended deadlock.

(4) Neither AIX 4.2 nor AIX 4.3 supports mutex attributes, requiring a NULL argument for the attr field in pthreads_mutex_init. The biggest practical consequence of this restricts each mutex to the threads of a single process, rather than sharing the mutex among several processes.

EXAMPLES.

Many texts illustrate mutex use superficially, but the most practical and explicit examples appear in LC's Pthreads Tutorial (URL: http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html) by Blaise Barney.

Condition Variables

SUMMARY.

A condition variable uses a shared variable to synchronize threads, as does a mutex (page 22), but with two key differences:

(1) the condition variable provides *value*-based access control, allowing a waiting thread to run only when a specified condition (perhaps on a counter or other sum) has been met, and

(2) it is active, not passive, automatically signalling the waiting thread(s) to awaken.

A condition variable is always bundled with a mutex variable to produce a condition-tested turn-taking combination, as follows:

One thread specifies the condition to be met (such as a counter threshold), locks an associated mutex, and calls the pthreads library primitive pthread_cond_wait. This routine then (a) unlocks the mutex for other threads to use, and (b) blocks the calling thread from running until it receives a wake-up signal. Other threads then take (mutex-enforced) turns locking/unlocking, doing their assigned work, and checking if the condition has been met with each new turn. When the condition is finally met, the thread that discovers this calls pthread_cond_signal to wake up the waiting thread to safely finish its own tasks (pthread_cond_broadcast will wake up many waiting threads simultaneously).

STRENGTHS.

The use of active signals with the condition-variable library routines eliminates costly "spin lock" polling overhead incurred when a mutex is used alone. And because the value of the condition variable, not merely the presence of a lock, controls thread behavior, this synchronization technique applies to many more cases than does binary control alone. Some strategists even suggest combining condition-variable tests with a timer routine to allow still greater flexibility: the waiting thread runs when either the condition is met or the time limit expires.

WEAKNESSES.

(1) Since a condition variable is always bundled with a mutex variable, the danger (noted above (page 22)) persists that miscoding the lock/unlock steps will yield a deadlock, especially if many pairs are used at once.

(2) A more serious problem is posed by multiple signals. Multiple threads in play at the same time can generate multiple wake-up signals (as multiple conditions are met). However, all but the first signal may easily be lost if the signalled thread happens by luck to already be awake when they arrive (yet all are needed to maintain proper thread synchronization). A general computational technique called semaphores (page 25), discussed in a later subsection, can be applied here to solve this lost-signal problem among parallel threads.

(3) Neither AIX 4.2 nor AIX 4.3 supports condition-variable attributes. The biggest practical consequence of this, as with mutex attributes, restricts each condition variable to the threads of a single process, rather than sharing it among several processes.

EXAMPLES.

Condition variables are widely mentioned but seldom thoroughly and carefully illustrated. The most thorough and practical examples again appear in LC's Pthreads Tutorial (URL: http://www.llnl.gov/computing/tutorials/workshops/workshop/pthreads/MAIN.html) by Blaise Barney.

Read/Write Lock

SUMMARY.

Consider the problem of efficiently managing threads that perform different kinds of shared-data accesses with markedly different frequencies. For example, a multithreaded process might need frequent data reads but only infrequent data writes. One solution is to build (from pthreads primitives) a customized composite synchronization tool that accommodates, even takes advantage of, the big difference in access rates. A "read/write lock" (or RWlock) thus combines one mutex with two separate condition variables. One condition variable counts reader threads, while the second counts writer threads. Readers swap access locks as often as possible with no real limits. Writers wait until the waiting-reader count reaches zero, when one is awakened to run and then awakens (with a broadcast) all newly waiting readers in turn.

STRENGTHS.

This special-purpose synchronization composite is built using only standard mutex and condition variable pthreads primitives (unlike semaphores (page 25), next subsection, a composite that calls for extra primitives). Read/write locks eliminate the polling overhead of mutex-only spin locks (page 22), and they also minimize the wait time for readers while still servicing the less frequent write requests. This shows how the extra work of building a "tuned" synchronization composite can improve overall thread performance.

WEAKNESSES.

(1) Such customized composites unfortunately rely on you (not some pretested, standardized library) to handle all the annoying details needed for practical code reliability: error handling, verifying lock ownership by checking thread IDs, and cleanup housekeeping from repeated uses.
(2) The absence of AIX support for mutex (page 22) and condition variable (page 23) attributes, noted in the previous subsections, is of course inherited here, with the same limitation to threads of a single process.

EXAMPLES.

An example read/write lock, coded except for the safety features mentioned above, appears with commentary in IBM's C and C++ Applications Development on AIX (URL: http://www.redbooks.ibm.com/abstracts/sg25674.html), SG245674, Chapter 5 (POSIX Threads), section 5.1.4.3 (abstracted online in HTML, but available fully only in PDF format).

Semaphores

SUMMARY.

Already noted as a weakness of condition-variable (page 23) use is the possibility that a race condition (page 18) for primary multithreaded tasks can reappear as a race condition for wake-up signals from multiple pthread_cond_wait calls. It is possible for an already awake and running thread to lose "extra" signals generated by multiple other threads trying to synchronize with it, and then halt later because these early wake ups are no longer available when they are really needed. A general programming solution (called semaphores) to this problem was originally developed by Dijkstra in 1968, and is now part of the standard computing science literature.

A "counting semaphore" can capture all incoming wake ups in a condition variable used as a task counter in a way that lets multiple threads synchronize without losing needed signals. Crucial for reliable success are two extra semaphore primitives, usually called UP and DOWN, that increment or decrement (respectively) the task counter and handle the associated thread blocking "atomically." This means that no other instruction can execute between the count change and the block (unlike a normal IF statement, which allows another thread to run between doing the test and carrying out the THEN/ELSE clause, possibly spoiling the test's relevance). The IFs usually used with condition variables and read/write locks are vulnerable to this problem in a multithreaded environment.

STRENGTHS.

The "atomic" nature of the semaphore instructions prevents wake-up signal "leakage" between count-change and thread-block steps, a known source of unintended deadlocks when many threads try to synchronize. Semaphores can thus manage waiting for task completion just as effectively as pthread joining (page 21) (above), but without the potential waste of storage space incurred by having many undetached (joinable) threads. For example, a semaphore that counts child threads could serve the same role (waiting for all of them to complete), with the same active-signal alert benefits, as joining a parent thread to all of those undetached children.

WEAKNESSES.

Like read/write locks, this is a composite thread synchronization technique that you need to build and customize from other pthreads primitives to meet the specific thread-coordination needs of your own program. Your coding and program testing will be more complex than if you rely on simple joining or mutex operations. Also, you will need to find or make a library of appropriate semaphore primitives (see EXAMPLES below).

EXAMPLES.

Semaphore design is a topic in many programming texts, and basic conceptual examples appear often in books on parallel programming and concurrency issues. IBM's own C and C++ Applications Development on AIX (URL: http://www.redbooks.ibm.com/abstracts/sg25674.html), SG245674, Chapter 5 (POSIX Threads), section 5.1.4.4 (abstracted online in HTML, but available fully only in PDF format), offers one relevant implementation. Tom Wagner and Don Towsely of the University of Massachusetts offer the complete source code for a C-language semaphore library intended for thread management in Appendix A of their 1995 report "Getting Started with POSIX Threads." (URL: http:www.cs.ucr.edu/~sshah/pthreads/tutorial.html) They also provide examples of its use.

Implementation Differences

Comparison Table

Despite the existence of pthreads standards, different vendors making different versions of different operating systems have implemented POSIX threads differently, sometimes with serious negative consequences for code portability. This table summarizes the known differences (except stack size, discussed below) most relevant to porting pthreads programs among the various LC computing systems (missing details are still being pursued and will be added when available).

Comparison of Vendor Implementations of POSIX Threads (at LC).

Pthreads Features	**IBM AIX 4.2**	**IBM AIX 4.3**	**Compaq Tru64 UNIX**	**Red Hat Linux**
Maximum threads/process	512			
Thread-waiting default	unjoinable, detached	joinable, undetached		joinable, undetached
Mutex attributes status	none (use NULL)	none (use NULL)		
Condition-variable attributes status	none (use NULL)	none (use NULL)		
Default contention scope	system (global)	process (local)		system (global)
Thread model	1:1 (kernel)	M:N (hybrid)		1:1 (kernel)

Both the default and the maximum allowed *stack size* for pthreads also vary greatly from one implementation to another on LC machines (and even within one operating system the differences are great).

Comparison of POSIX Threads Stack Size Limits (at LC).

LC Machine	**Operating System**	**Chip Type**	**Default Stack (MB)**	**Maximum Stack (MB)**
Lilac	Chaos	IA32	67.1	1072 (2 threads) 92 (32 threads)
Thunder	Chaos	IA64	32.6	25000
MCR	Chaos	IA32	2.1	1072 (2 threads) 92 (32 threads)
UM, UV	AIX	Power4	0.196	260
White	AIX	Power3	0.098	260

These great variations make relying on the default pthreads stack size impractical, especially for codes intended to be portable. Instead, set your own stack size explicitly:

- Use library routine

  ```
  pthread_attr_setstacksize (&attr, stacksize)
  ```

  to specify overtly your desired pthreads stack (up to the maximum value shown above for the target machine).
- On (Red Hat) Linux/CHAOS systems at LC, always allocate at least *twice* the actual amount of memory that you want for each thread stack (so to handle 20 MB of data, for example, allocate a stack of 41 MB). This quirk is a side effect of how Red Hat Linux on IA chips splits the stack into two parts.

General Linux Support

Because of the growing importance of Linux (CHAOS/SLURM) systems at LC, users who plan to develop POSIX threads codes under Linux may want specialized advice relevant to that environment. If you are just starting to use pthreads, Mark Hays's

```
POSIX Threads Tutorial
http://www.math.arizona.edu/swig/pthreads/threads.html
```

covers the usual issues with examples and techniques all tested exclusively on Linux systems (but not on the local CHAOS version). On the other hand, if you are converting pthreads code from elsewhere to use the (former) LinuxThreads library, then Xavier LeRoy's

```
The LinuxThreads Library
http://pauillac.inria.fr/~xleroy/linuxthreads/
```

may prove more useful because it cites and links to many comparative sources on the GLIBC compiler's support for pthreads, on the threads-aware GDB debugger, and on LinuxThreads frequently asked questions. (But also see the comparison with the newer, replacement Native POSIX Threading Library in the next section (page 30).)

MKL is Intel's threaded Math Kernel Library, a good source for BLAS and LAPACK routines in the Linux environment.

MKL is available *only* on LC machines with Intel chips, that is, only on the Intel Linux clusters (such as ILX, MCR, and ALC on the open network, or on Adelie or Emperor on the secure network). The relevant Linux system subdirectories are:

```
/usr/local/intel/mkl/lib/32   (the library files)
                     /include  (the include files)
                     /doc      (vendor documentation)
```

The Linux/CHAOS environment variable OMP_NUM_THREADS controls the number of threads spawned by the MKL routines (by default, MKL sets the number of threads equal to the number of processors where you run).

More generally, local resource manager SLURM's job-control user utility, SRUN, offers several options intended to help you properly assign compute resources to pthreads applications. SRUN's -c (lowercase) option lets you specify multiple CPUs per task, while -T (uppercase) lets you specify the number of threads per job. See the SRUN sections of the SLURM Reference Manual (URL: http://www.llnl.gov/LCdocs/slurm) for details on the complex ways in which SRUN options (these two and others) can interact.

Native POSIX Threading Library

All versions of the Linux kernel prior to version 2.6 supported threads by using the LinuxThreads library. Although historically *called* a POSIX threads library and compiled to a file called libpthread.a, LinuxThreads always had noteworthy thread-support limitations:
(a) It used a *compile*-time setting for the number of threads that a single process could create.
(b) It used a fairly high-overhead "manager thread" to coordinate (create, destroy) all the threads owned by any one process.
(c) It handled signals on a per-process rather than a per-thread basis.
(d) It lacked synchronization primitives for effectively communicating between threads and for sharing resources.

Starting with Linux kernel version 2.6 (which at LC means starting with the deployment of CHAOS 2.0 across local Linux clusters in the summer of 2004), LinuxThreads was replaced by an alternative called the Native POSIX Threading Library (NPTL, sometimes also surprisingly abbreviated as NTPL!). NPTL was developed as a closely integrated part of the whole Linux run-time package, so it offers high-performance POSIX thread support. It works well for high-capacity or high-load threaded applications and it complements GLIBC. CHAOS 2.0 makes NPTL available to LC's Linux users.

Therefore, the rest of this section spells out some important differences between the older, less effective LinuxThreads library and the newer, more effective Native POSIX Threading Library:

- AVAILABILITY.
  To confirm that the Linux machine on which you are running really offers NPTL, use the GETCONF command to query the GNU_LIBPTHREAD_VERSION environment variable (uppercase):

  ```
  getconf GNU_LIBPTHREAD_VERSION
  ```

  A reply of "NPTL 0.60" confirms the new library, while a reply of "linuxthreads-0.10" reveals that the old library is still the default.

- ENABLING NPTL.
  When building a C library, such as GLIBC, you enable Native POSIX Threading Library support by invoking the cofiguration option

  ```
  --enable-add-ons=nptl
  ```

  (where the alternative argument "linuxthreads" would have enabled the older library instead).

- CODE SIZE.
  In general, using NPTL will significantly *increase* the size of your code after you recompile. You may want to seek out additional optimization options to compensate for this size increase if resource constraints are important for you. You might want to try general (-f) rather than machine-specific (-m) optimizations, or try relaxing strict aliasing (-fno-strict-aliasing).

- EFFICIENCY EFFECTS.
  NPTL generally performs much better than LinuxThreads. NPTL is far more compliant with the POSIX standard, and it supports mutexes that are shared among threads (for simplier resource sharing and greater parallelism). Some commercial benchmark metrics show NPTL applications to be as much as three orders of magnitude faster than the same code using LinuxThreads.
- SIGNAL SUPPORT.
  NPTL generally supports POSIX signals and signal handling much better than LinuxThreads (which used generic UNIX signals). With NPTL you can send signals from one thread to another thread, not merely from one process to another process (although between-process signals often perform better as well). Also, NPTL lets you transfer information from one thread to another using signal *arguments*, a communication technique not possible under LinuxThreads.
- THREAD IDENTIFICATION.
  Under LinuxThreads, each thread had a unique process ID (PID), returned by the GETPID function. Under NPTL, each thread has its own unique thread ID (TID) because every thread *shares* the process ID (PID) of its parent process. So you can no longer use GETPID to distinguish among separate threads of the same process. The exec() function under NPTL is thread-aware (so that every thread can inherit the PID of its caller). And likewise, under NPTL, pthread_at_fork() no longer works the same as vfork().
- MANAGER THREAD.
  To minimize administrative overhead and simplify the process/thread relationship, NPTL eliminates the "manager thread" formerly used under LinuxThreads. You may need to change your application code, however, if you relied on the manager thread to count the number of currently running threads or to receive any signals.
- OBSOLETE FUNCTIONS.
  Some functions unique to LinuxThreads are no longer available (or needed) under the Native POSIX Threading Library. For example,

  ```
  pthread_kill_other_threads_np()
  ```

  is gone, because LinuxThreads needed it to simulate the POSIX-conformant behavior of exec() that NPTL provides as a native feature.

Pthreads Performance Benchmarks

SPHINX:

LLNL's Center for Applied Scientific Computing (CASC) now provides as publicly downloadable code a C-language integrated parallel microbenchmark suite to conduct performance tests of pthreads on many different platforms (Compaq, IBM, SGI, and Sun). LLNL's pthreads benchmark suite, called Sphinx, has these features:

- Accesses each test action (such as message pingpong) through a function pointer, allowing different threads or tasks to execute different functions at once. This supports measurement of highly complex parallel actions.
- Times repeated calls (iterations) of each test action, stopping either when the standard deviation of the repetitions becomes less than a user-specified percentage of their mean or, if that never happens, after a user-specified maximum number of repetitions.
- (Optionally) corrects for test-suite ("harness") overhead and automatically warns if that overhead exceeds the measurement value of the test.
- Is highly portable (vendor-dependent binding of threads to processors is the chief threat to Sphinx portability).
- Covers (with suitable adaptations) pthreads, MPI, and OpenMP performance testing. Documentation at the Sphinx web site (below) specifically explains which tests apply to which features of the three approaches to parallelization.

Sphinx is available to the public at this open URL:

`http://www.llnl.gov/CASC/sphinx`

This Sphinx web site (UCLR-CODE-99026) provides all needed usage information and relevant files, including:

- A descriptive inventory of every file in the current Sphinx distribution, which includes every input file for the ASCI milepost tests.
- Build and execution instructions for the test suite.
- Input file format and the input modes that Sphinx accepts.
- Output file format and the four Sphinx output streams.
- Specific test descriptions and allowed independent variables.
- References to (and in some cases even the full text of) published papers that present and discuss Sphinx results.

IBM PERFORMANCE PROBLEMS:

As of April, 2002, LC's massively parallel IBM computers sometimes showed significant performance problems for MPI programs, especially for those programs that (heavily) use library routines MPI_ALLREDUCE or MPI_BARRIER. One production physics code where MPI_ALLREDUCE was algorithmically expected to use about 1.9% of the total run time, for example, actually spent 90% of its MPI time and 30% of its physics time just executing MPI_ALLREDUCE. Extensive comparative testing by LC staff members suggests that the following kinds of codes are most susceptible to these serious performance problems:

- Parallel codes with fine-grained parallelization,
- Hybrid codes that combine MPI with OpenMP or with POSIX threads (Pthreads), and
- Codes that make heavy use of MPI_ALLREDUCE or MPI_BARRIER.

Users who see (or who wish to avoid) these problems are urged to profile their codes by running /usr/local/mpiP and to read the "IBM Confidential" analysis available at (OCF, special password required, request from the LC Hotline):

`http://www-r.llnl.gov/icc/viewgraphs/viewgraphs02/apr02/jones/index.htm`

for more details on the underlying cause of these performance problems and for a few suggestions to work around them.

Disclaimer

Keyword Index

To see an alphabetical list of keywords for this document, consult the next section (page 36).

Keyword	Description
entire	This entire document.
title	The name of this document.
scope	Topics covered in this document.
availability	Where these programs run.
who	Who to contact for assistance.
introduction	Topics and issues covered.
definitions	Defines thread features, terms.
analogies	Alternative ways to use threads.
benefits	Advantages of pthreads use.
models	Three ways systems implement threads.
environment-variables	AIX threads-control e-vars.
tuning	AIX threads-control e-vars.
dangers	Known problems of ptheads use.
synchronization	Thread synchronization techniques.
synchronization-chart	Tabular technique comparison.
synchronization-alternatives	Five ways to synchronize threads.
joining	Joining to synchronize threads.
mutex	Mutex variable to synchronize threads.
condition	Condition var to synchronize threads.
rwlock	Read/write lock to synchronize threads.
semaphores	Semaphores to synchronize threads.
differences	Vendor variations in pthreads details.
comparison-table	Pthreads implementations compared.
linux-pthreads	General issues about Linux pthreads.
nptl-pthreads	Native POSIX Threading Lib vs. LinuxThreads.
linuxthreads	Native POSIX Threading Lib vs. LinuxThreads.
benchmarks	Microbenchmark suite for pthreads.
index	The structural index of keywords.
a	The alphabetical index of keywords.
date	The latest changes to this document.
revisions	The complete revision history.

Alphabetical List of Keywords

Keyword	Description
a	The alphabetical index of keywords.
analogies	Alternative ways to use threads.
availability	Where these programs run.
benchmarks	Microbenchmark suite for pthreads.
benefits	Advantages of pthreads use.
comparison-table	Pthreads implementations compared.
condition	Condition var to synchronize threads.
dangers	Known problems of ptheads use.
date	The latest changes to this document.
definitions	Defines thread features, terms.
differences	Vendor variations in pthreads details.
entire	This entire document.
environment-variables	AIX threads-control e-vars.
index	The structural index of keywords.
introduction	Topics and issues covered.
joining	Joining to synchronize threads.
linuxthreads	Native POSIX Threading Lib vs. LinuxThreads.
linux-pthreads	General issues about Linux pthreads.
models	Three ways systems implement threads.
mutex	Mutex variable to synchronize threads.
nptl-pthreads	Native POSIX Threading Lib vs. LinuxThreads.
revisions	The complete revision history.
rwlock	Read/write lock to synchronize threads.
scope	Topics covered in this document.
semaphores	Semaphores to synchronize threads.
synchronization	Thread synchronization techniques.
synchronization-alternatives	Five ways to synchronize threads.
synchronization-chart	Tabular technique comparison.
title	The name of this document.
tuning	AIX threads-control e-vars.
who	Who to contact for assistance.

Date and Revisions

Revision Date	Keyword Affected	Description of Change
25May05	comparison-table	New stack size comparison added.
07Jul04	nptl-pthreads	New section on native threads library.
	differences	Divided into three subsections.
	index	New keywords for new sections.
27Oct03	introduction	Cross ref to SLURM manual added.
	environment-variables	Relevant SLURM environment vars. added.
	differences	Relevant SRUN options noted.
18Jun03	differences	MKL threaded Linux library noted.
07May02	benchmarks	MPI/pthreads performance problem.
12Nov01	introduction	Linux support noted.
	differences	Linux details, cross refs added.
11Jun01	benchmarks	New section on Sphinx added.
	index	New keyword for new section.
16Feb00	entire	First edition of LC PTHREADS manual.

TRG (25May05)

UCRL-WEB-201477

TRG (25May05) Contact on the OCF: lc-hotline@llnl.gov, on the SCF: lc-hotline@pop.llnl.gov