Covering Multithreading Basics 1

The word *multithreading* can be translated as *multiple threads of control* or *multiple flows of control*. While a traditional UNIX process always has contained and still does contain a single thread of control, multithreading (MT) separates a process into many execution threads, each of which runs independently.

Multithreading your code can

- Improve application responsiveness
- Use multiprocessors more efficiently
- Improve program structure
- Use fewer system resources

This chapter explains some multithreading terms, benefits, and concepts. If you are ready to start using multithreading, skip to the chapter "Basic Threads Programming" on page 11.

Defining Multithreading Terms

Table 1-1 introduces some of the terms used in this book.

Table 1-1 Multithreading Terms

Term	Definition
Process	The UNIX environment (context like file descriptors, user ID, and so on) created with the `fork`(2) system call, which is set up to run a program.
Thread	A sequence of instructions executed within the context of a process.
pthreads (POSIX threads)	A POSIX 1003.1c compliant threads interface.
Solaris threads	A SunSoft™ threads interface that is not POSIX compliant. A predecessor of pthreads.
Single-threaded	Restricting access to a single thread.
Multithreaded	Allowing access to two or more threads.
User- or Application-level threads	Threads managed by the threads library routines in user (as opposed to kernel) space.
Lightweight processes	Threads in the kernel that execute kernel code and system calls (also called LWPs).
Bound threads	Threads that are permanently bound to LWPs.
Unbound threads	A default Solaris thread that context switches very quickly without kernel support.
Attribute object	Contains opaque data types and related manipulation functions used to standardize some of the configurable aspects of POSIX threads, mutexes, and condition variables.
Mutual exclusion locks	Functions that lock and unlock access to shared data
Condition variables	Functions that block threads until a change of state.
Counting semaphore	A memory-based synchronization mechanism.
Parallelism	A condition that arises when at least two threads are *executing* simultaneously.
Concurrency	A condition that exists when at least two threads are *making progress*. A more generalized form of parallelism that can encompass time-slicing as a form of virtual parallelism.

Meeting Multithreading Standards

The concept of multithreaded programming goes back to at least the 1960s. Its development on UNIX systems goes back to the mid-1980s. While there is agreement about what multithreading is and the features necessary to support it, the interfaces used to implement multithreading have varied greatly.

For several years a group called POSIX (Portable Operating System Interface) 1003.4a has been working on standards for multithreaded programming. The standard has now been ratified. This guide is based on the POSIX standards: P1003.1b final draft 14 (realtime), and P1003.1c final draft 10 (multithreading).

This book now covers both POSIX threads (also called *pthreads*) and Solaris threads. Solaris threads were available in the Solaris 2.4 release, and are not functionally different from POSIX threads. However, because POSIX threads are more portable than Solaris threads, this book covers multithreading from the POSIX perspective. Subjects specific to Solaris threads, only, are covered in the chapter "Programming with Solaris Threads."

Benefiting From Multithreading

Improving Application Responsiveness

Any program in which many activities are not dependent upon each other can be redesigned so that each activity is defined as a thread. For example, the user of a multithreaded GUI does not have to wait for one activity to complete before starting another.

Using Multiprocessors Efficiently

Typically, applications that express concurrency requirements with threads need not take into account the number of available processors. The performance of the application improves transparently with additional processors.

Numerical algorithms and applications with a high degree of parallelism, such as matrix multiplications, can run much faster when implemented with threads on a multiprocessor.

Improving Program Structure

Many programs are more efficiently structured as multiple independent or semi-independent units of execution instead of as a single, monolithic thread. Multithreaded programs can be more adaptive to variations in user demands than single threaded programs.

Using Fewer System Resources

Programs that use two or more processes that access common data through shared memory are applying more than one thread of control.

However, each process has a full address space and operating systems state. The cost of creating and maintaining this large amount of state makes each process much more expensive than a thread in both time and space.

In addition, the inherent separation between processes can require a major effort by the programmer to communicate between the threads in different processes or to synchronize their actions.

Combining Threads and RPC

By combining threads and a remote procedure call (RPC) package, you can exploit nonshared-memory multiprocessors (such as a collection of workstations). This combination distributes your application relatively easily and treats the collection of workstations as a multiprocessor.

For example, one thread might create child threads. Each of these children could then place a remote procedure call, invoking a procedure on another workstation. Although the original thread has merely created threads that are now running in parallel, this parallelism involves other computers.

Understanding Basic Multithreading Concepts

Concurrency and Parallelism

In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution.

In the same multithreaded process in a shared-memory multiprocessor environment, each thread in the process can run on a separate processor at the same time, resulting in parallel execution.

When the process has as many threads as, or fewer threads than, there are processors, the threads support system and the operating system ensure that each thread runs on a different processor.

For example, in a matrix multiplication that has the same number of threads and processors, each thread (and each processor) computes a row of the result.

Looking at Multithreading Structure

Traditional UNIX already supports the concept of threads—each process contains a single thread, so programming with multiple processes is programming with multiple threads. But a process is also an address space, and creating a process involves creating a new address space.

Creating a thread is much less expensive when compared to creating a new process, because the newly created thread uses the current process address space. The time it takes to switch between threads is much less than the time it takes to switch between processes, partly because switching between threads does not involve switching between address spaces.

Communicating between the threads of one process is simple because the threads share everything—address space, in particular. So, data produced by one thread is immediately available to all the other threads.

The interface to multithreading support is through a subroutine library, `libpthread` for POSIX threads, and `libthread` for Solaris threads. Multithreading provides flexibility by decoupling kernel-level and user-level resources.

User-level Threads

Threads are the primary programming interface in multithreaded programming. User-level threads[1] are handled in user space and avoid kernel context switching penalties. An application can have hundreds of threads and still not consume many kernel resources. How many kernel resources the application uses is largely determined by the application.

Threads are visible only from within the process, where they share all process resources like address space, open files, and so on. The following state is unique to each thread.

- Thread ID
- Register state (including PC and stack pointer)
- Stack
- Signal mask
- Priority
- Thread-private storage

Because threads share the process instructions and most of the process data, a change in shared data by one thread can be seen by the other threads in the process. When a thread needs to interact with other threads in the same process, it can do so without involving the operating system.

By default, threads are very lightweight. But, to get more control over a thread (for instance, to control scheduling policy more), the application can bind the thread. When an application binds threads to execution resources, the threads become kernel resources (see "System Scope (Bound Threads)" on page 8 for more information).

To summarize, user-level threads are:

- Inexpensive to create because they do not need to create their own address space. They are bits of virtual memory that are allocated from your address space at run time.
- Fast to synchronize because synchronization is done at the application level, not at the kernel level.
- Easily managed by the threads library, `libpthread` or `libthread`.

1. User-level threads are named to distinguish them from kernel-level threads, which are the concern of systems programmers, only. Because this book is for application programmers, kernel-level threads are not discussed.

Lightweight Processes

The threads library uses underlying threads of control called *lightweight processes* that are supported by the kernel. You can think of an LWP as a virtual CPU that executes code or system calls.

You usually do not need to concern yourself with LWPs to program with threads. The information here about LWPs is provided as background, so you can understand the differences in scheduling scope, described on page 8.

Note – The LWPs in the Solaris 2.x system are *not* the same as the LWPs in the SunOS™ 4.0 LWP library, which are not supported in the Solaris 2.x system.

Much as the `stdio` library routines such as `fopen(3S)` and `fread(3S)` use the `open(2)` and `read(2)` functions, the threads interface uses the LWP interface, and for many of the same reasons.

Lightweight processes (LWPs) bridge the user level and the kernel level. Each process contains one or more LWPs, each of which runs one or more user threads. The creation of a thread usually involves just the creation of some user context, but not the creation of an LWP.

Each LWP is a kernel resource in a kernel pool, and is allocated (attached) and de-allocated (detached) to a thread on a per thread basis. This happens as threads are scheduled or are created and destroyed.

Scheduling

POSIX specifies three scheduling policies: first-in-first-out (`SCHED_FIFO`), round-robin (`SCHED_RR`), and custom (`SCHED_OTHER`). `SCHED_FIFO` is a queue-based scheduler with different queues for each priority level. `SCHED_RR` is like FIFO except that each thread has a execution time quota.

Both `SCHED_FIFO` and `SCHED_RR` are POSIX Realtime extensions. `SCHED_OTHER` is the default scheduling policy.

See “LWPs and Scheduling Classes” on page 127“ for information about the `SCHED_OTHER` policy, and about emulating some properties of the POSIX `SCHED_FIFO` and `SCHED_RR` policies.

Two scheduling scopes are available: process scope for unbound threads and system scope for bound threads. Threads with differing scope states can coexist on the same system and even in the same process. In general, the scope sets the range in which the threads policy is in effect.

Process Scope (Unbound Threads)

Unbound threads are created `PTHREAD_SCOPE_PROCESS` and are private to a process.These threads are scheduled in user space to attach and detach from available LWPs in the LWP pool.

In most cases, threads should be `PTHREAD_SCOPE_PROCESS`. This allows the threads to float among the LWPs, and this improves threads performance (and is equivalent to creating a Solaris thread in the `THR_UNBOUND` state).

System Scope (Bound Threads)

Bound threads are created `PTHREAD_SCOPE_SYSTEM`. A thread with a scope of `PTHREAD_SCOPE_SYSTEM` is global to the system.

Each bound thread is bound to an LWP for the lifetime of the thread. This is equivalent to creating a Solaris thread in the `THR_BOUND` state. You can bind a thread to give it an alternate signal stack or to use special scheduling attributes with Realtime scheduling.

Cancellation

Thread cancellation allows a thread to terminate the execution of any other thread in the process. The target thread (the one being cancelled) can keep cancellation requests pending and can perform application-specific cleanup when it acts upon the cancellation notice.

The pthreads cancellation feature permits either asynchronous or deferred termination of a thread. Asynchronous cancellation can occur at any time; deferred cancellation can occur only at defined points. Deferred cancellation is the default type.

Synchronization

Synchronization allows you to control program flow and access to shared data for concurrently executing threads.

The three synchronization models are mutex locks, condition variables, and semaphores.

- Mutex locks allow only one thread at a time to execute a specific section of code, or to access specific data.
- Condition variables block threads until a particular condition is true.
- Counting semaphores typically coordinate access to resources. The count is the limit on how many threads can have access to a semaphore. When the count is reached, the semaphore blocks.

1