Multithreaded Programming Guide

Beta

Part No: E35303
October 2012

121010@25097

Contents

Preface 13

1 **Covering Multithreading Basics** 17

Multithreading Terms 17

Oracle Solaris Multithreading Libraries and Standards 19

Benefiting From Multithreading 20

Improving Application Responsiveness 20

Using Multiprocessors Efficiently 20

Improving Program Structure 20

Using Fewer System Resources 21

Combining Threads and RPC 21

Multithreading Concepts 21

Concurrency and Parallelism 21

Multithreading Structure 22

Thread Scheduling 23

Thread Cancellation 23

Thread Synchronization 24

Using the 64-bit Architecture 24

2 **Basic Threads Programming** 27

Lifecycle of a Thread 27

The Pthreads Library 28

Creating a Default Thread 28

Waiting for Thread Termination 30

Simple Threads Example 31

Detaching a Thread 32

Creating a Key for Thread-Specific Data 33

Deleting the Thread-Specific Data Key ... 34
Setting Thread-Specific Data ... 35
Getting Thread-Specific Data ... 36
Global and Private Thread-Specific Data Example ... 36
Getting the Thread Identifier ... 38
Comparing Thread IDs ... 39
Calling an Initialization Routine for a Thread ... 39
Yielding Thread Execution ... 40
Setting the Thread Policy and Scheduling Parameters ... 40
Getting the Thread Policy and Scheduling Parameters ... 41
Setting the Thread Priority ... 42
Sending a Signal to a Thread ... 43
Accessing the Signal Mask of the Calling Thread ... 43
Forking Safely ... 44
Terminating a Thread ... 45
Finishing Up ... 45
Cancel a Thread ... 46
Cancelling a Thread ... 47
Enabling or Disabling Cancellation ... 48
Setting Cancellation Type ... 48
Creating a Cancellation Point ... 49
Pushing a Handler Onto the Stack ... 49
Pulling a Handler Off the Stack ... 50

3 Thread Attributes ... 51
Attribute Object ... 51
Initializing Attributes ... 52
Destroying Attributes ... 54
Setting Detach State ... 54
Getting the Detach State ... 55
Setting the Stack Guard Size ... 56
Getting the Stack Guard Size ... 57
Setting the Scope ... 57
Getting the Scope ... 58
Setting the Thread Concurrency Level ... 59

Getting the Thread Concurrency Level 59
Setting the Scheduling Policy 60
Getting the Scheduling Policy 61
Setting the Inherited Scheduling Policy 62
Getting the Inherited Scheduling Policy 62
Setting the Scheduling Parameters 63
Getting the Scheduling Parameters 64
About Stacks 65
Setting the Stack Size 66
Getting the Stack Size 67
Setting the Stack Address and Size 68
Getting the Stack Address and Size 69

4 Programming with Synchronization Objects 71
Mutual Exclusion Lock Attributes 72
Initializing a Mutex Attribute Object 73
Destroying a Mutex Attribute Object 75
Setting the Scope of a Mutex 75
Getting the Scope of a Mutex 76
Setting the Mutex Type Attribute 76
Getting the Mutex Type Attribute 77
Setting the Mutex Attribute's Protocol 78
Getting the Mutex Attribute's Protocol 80
Setting the Mutex Attribute's Priority Ceiling 80
Getting the Mutex Attribute's Priority Ceiling 81
Setting the Mutex's Priority Ceiling 82
Getting the Mutex's Priority Ceiling 82
Setting the Mutex's Robust Attribute 83
Getting the Mutex's Robust Attribute 84
Using Mutual Exclusion Locks 85
Initializing a Mutex 86
Making a Mutex Consistent 87
Locking a Mutex 88
Unlocking a Mutex 89
Locking a Mutex Without Blocking 90

- Locking a Mutex Before a Specified Absolute Time ... 91
- Locking a Mutex Within a Specified Time Interval ... 92
- Destroying a Mutex ... 93
- Code Examples of Mutex Locking ... 93

Using Spin Locks ... 98
- Initializing a Spin Lock ... 99
- Acquiring a Spin Lock ... 100
- Acquiring a Non-Blocking Spin Lock ... 100
- Unlocking a Spin Lock ... 101
- Destroying a Spin Lock ... 101

Condition Variable Attributes ... 102
- Initializing a Condition Variable Attribute ... 103
- Removing a Condition Variable Attribute ... 104
- Setting the Scope of a Condition Variable ... 104
- Getting the Scope of a Condition Variable ... 105
- Setting the Clock Selection Condition Variable ... 106
- Getting the Clock Selection Condition Variable ... 106

Using Condition Variables ... 107
- Initializing a Condition Variable ... 108
- Blocking on a Condition Variable ... 109
- Unblocking One Thread ... 110
- Blocking Until a Specified Time ... 111
- Blocking For a Specified Interval ... 113
- Unblocking All Threads ... 114
- Destroying the Condition Variable State ... 115
- Lost Wake-Up Problem ... 115
- Producer and Consumer Problem ... 116

Synchronization With Semaphores ... 119
- Named and Unnamed Semaphores ... 120
- Counting Semaphores Overview ... 120
- Initializing a Semaphore ... 121
- Incrementing a Semaphore ... 122
- Blocking on a Semaphore Count ... 123
- Decrementing a Semaphore Count ... 124
- Destroying the Semaphore State ... 124
- Producer and Consumer Problem Using Semaphores ... 125

Read-Write Lock Attributes 126
Initializing a Read-Write Lock Attribute 127
Destroying a Read-Write Lock Attribute 127
Setting a Read-Write Lock Attribute 128
Getting a Read-Write Lock Attribute 128
Using Read-Write Locks 129
Initializing a Read-Write Lock 129
Acquiring the Read Lock on Read-Write Lock 130
Acquiring a Read Lock on a Read-Write Lock Before a Specified Absolute Time 131
Acquiring a Non-Blocking Read Lock on a Read-Write Lock 132
Acquiring the Write Lock on a Read-Write Lock 133
Acquiring a Non-blocking Write Lock on a Read-Write Lock 134
Acquiring a Write Lock on a Read-Write Lock Before a Specified Absolute Time 134
Unlocking a Read-Write Lock 135
Destroying a Read-Write Lock 136
Using Barrier Synchronization 137
Initializing a Synchronization Barrier 137
Waiting for Threads to Synchronize at a Barrier 138
Destroying a Synchronization Barrier 139
Initializing a Barrier Attributes Object 139
Setting a Barrier Process-Shared Attribute 140
Getting a Barrier Process-Shared Attribute 140
Destroying a Barrier Attributes Object 141
Synchronization Across Process Boundaries 142
Producer and Consumer Problem Example 142
Comparing Primitives 143

5 Programming With the Oracle Solaris Software 145
Forking Issues in Process Creation 145
Fork-One Model 146
Fork-All Model 148
Choosing the Right Fork 149
Process Creation: `exec` and `exit` Issues 149
Timers, Alarms, and Profiling 149
Timers 149

Alarms 150
Profiling a Multithreaded Program 150
Nonlocal Goto: `setjmp` and `longjmp` 150
Resource Limits 151
LWPs and Scheduling Classes 151
Timeshare Scheduling 152
Realtime Scheduling 152
Fair Share Scheduling 153
Fixed Priority Scheduling 153
Extending Traditional Signals 153
Synchronous Signals 154
Asynchronous Signals 154
Continuation Semantics 155
Operations on Signals 155
Thread-Directed Signals 157
Completion Semantics 158
Signal Handlers and Async-Signal Safety 159
Interrupted Waits on Condition Variables 161
I/O Issues 162
I/O as a Remote Procedure Call 162
Tamed Asynchrony 162
Asynchronous I/O 162
Shared I/O and New I/O System Calls 164
Alternatives to `getc` and `putc` 164
New System Calls For Reliable Multithreaded Programming 165

6 Programming With Oracle Solaris Threads 167
Comparing APIs for Oracle Solaris Threads and POSIX Threads 167
Major API Differences 168
Function Comparison Table 168
Unique Oracle Solaris Threads Functions 172
Suspending Thread Execution 172
Continuing a Suspended Thread 173
Similar Synchronization Functions: Read-Write Locks 173
Initialize a Read-Write Lock 174

- Acquiring a Read Lock 175
- Trying to Acquire a Read Lock 176
- Acquiring a Write Lock 176
- Trying to Acquire a Write Lock 177
- Unlock a Read-Write Lock 178
- Destroying the Read-Write Lock State 178

Similar Oracle Solaris Threads Functions 179
- Creating a Thread 180
- Getting the Minimal Stack Size 182
- Acquiring the Thread Identifier 183
- Yield Thread Execution 183
- Send a Signal to a Thread 184
- Access the Signal Mask of the Calling Thread 184
- Terminate a Thread 185
- Wait for Thread Termination 185
- Creating a Thread-Specific Data Key 187
- Setting the Thread-Specific Data Value 187
- Getting the Thread-Specific Data Value 188
- Set the Thread Priority 188
- Get the Thread Priority 189

Similar Synchronization Functions: Mutual Exclusion Locks 190
- Initialize a Mutex 190
- Destroy a Mutex 192
- Acquiring a Mutex 192
- Releasing a Mutex 193
- Trying to Acquire a Mutex 193

Similar Synchronization Functions: Condition Variables 194
- Initialize a Condition Variable 194
- Destroying a Condition Variable 195
- Waiting for a Condition 196
- Wait for an Absolute Time 196
- Waiting for a Time Interval 197
- Unblock One Thread 198
- Unblock All Threads 198

Similar Synchronization Functions: Semaphores 199
- Initialize a Semaphore 199

Increment a Semaphore 200
Block on a Semaphore Count 201
Decrement a Semaphore Count 201
Destroy the Semaphore State 202
Synchronizing Across Process Boundaries 202
Example of Producer and Consumer Problem 202
Special Issues for `fork()` and Oracle Solaris Threads 204

7 Safe and Unsafe Interfaces 205
Thread Safety 205
MT Interface Safety Levels 206
Reentrant Functions for Unsafe Interfaces 207
Async-Signal-Safe Functions in Oracle Solaris Threads 208
MT Safety Levels for Libraries 209
Unsafe Libraries 209

8 Compiling and Debugging 211
Setting Up the Oracle Solaris Environment for Developing Multithreaded Applications 211
Compiling a Multithreaded Application 211
Preparing for Compilation 211
Choosing Oracle Solaris or POSIX Threads 212
Including `<thread.h>` or `<pthread.h>` 212
Compiling and Linking a Multithreaded Program 213
Linking With `-lrt` for POSIX Semaphores 214
Alternate Threads Library 214
Debugging a Multithreaded Program 214
Common Oversights in Multithreaded Programs 214
Tracing and Debugging with DTrace 215
Profiling with Performance Analyzer 216
Detecting Data Races and Deadlocks Using Thread Analyzer 216
Using `dbx` 216
Tracing and Debugging With the TNF Utilities 218
Using `truss` 218
Using `mdb` 218

9 Programming Guidelines 221
Rethinking Global Variables 221
Providing for Static Local Variables 222
Synchronizing Threads 223
Single-Threaded Strategy 224
Reentrant Function 224
Avoiding Deadlock 226
Deadlocks Related to Scheduling 227
Locking Guidelines 227
Finding Deadlocks 227
Some Basic Guidelines for Threaded Code 228
Creating and Using Threads 229
Working With Multiprocessors 229
Underlying Architecture 230
Examples of Threads Programs 234
Further Reading 234

A Extended Example: A Thread Pool Implementation 235
What is a Thread Pool? 235
About the Thread Pool Example 236
Thread Pool Functions 236
Thread Pool Code Examples 237
What the Thread Pool Example Shows 246

Index 247

Preface

The *Multithreaded Programming Guide* describes the multithreaded programming interfaces for POSIX threads and Oracle Solaris threads in the Oracle Solaris Operating System (Oracle Solaris OS). This guide shows application programmers how to create new multithreaded programs and how to add multithreading to existing programs.

Although this guide covers both the POSIX and Oracle Solaris threads interfaces, most topics assume a POSIX threads interest. Information applying to only Oracle Solaris threads is covered in Chapter 6, "Programming With Oracle Solaris Threads." The two sets of interfaces share a common implementation and are fully compatible with one another. Calls to POSIX threads interfaces can be freely intermixed with calls to Oracle Solaris threads interfaces.

POSIX threads information can be found in the Single UNIX Specification Version 3 at `http://www.opengroup.org/`.

Note – This Oracle Solaris release supports systems that use the SPARC and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, Pentium, and Xeon EM64T. For a list of supported systems see the *Oracle Solaris OS: Hardware Compatibility Lists*. This document cites any implementation differences between the platform types.

In this document the term "x86" refers to 64-bit and 32-bit systems manufactured using processors compatible with the AMD64 or Intel Xeon/Pentium product families. For supported systems, see the *Oracle Solaris 10 Hardware Compatibility List*.

Who Should Use This Book

This guide is for application developers who want to create new multithreaded programs or add multithreading to existing programs.

Developers that use this book should be familiar with and be able to use the following technologies:

- A UNIX SVR4 system - preferably the current Oracle Solaris release.
- The C programming language - multithreading interfaces are provided by the standard C library.

- The principles of concurrent or parallel programming (as opposed to sequential programming).

How This Guide Is Organized

Chapter 1, "Covering Multithreading Basics," gives a structural overview of threads implementation in this release.

Chapter 2, "Basic Threads Programming," discusses the general POSIX threads routines, emphasizing creating a thread with default attributes.

Chapter 3, "Thread Attributes," covers creating a thread with nondefault attributes.

Chapter 4, "Programming with Synchronization Objects," covers the threads synchronization routines.

Chapter 5, "Programming With the Oracle Solaris Software," discusses changes to the operating environment to support multithreading.

Chapter 6, "Programming With Oracle Solaris Threads," covers Oracle Solaris threads (as opposed to POSIX threads) interfaces.

Chapter 7, "Safe and Unsafe Interfaces," covers multithreading safety issues.

Chapter 8, "Compiling and Debugging," covers the basics of compiling and debugging multithreaded applications.

Chapter 9, "Programming Guidelines," discusses issues that affect programmers writing multithreaded applications.

Appendix A, "Extended Example: A Thread Pool Implementation," shows how to implement a pool of worker threads.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Related Books

Multithreading requires a different way of thinking about function interactions. The following books are recommended reading.

- *Multicore Application Programming: for Windows, Linux, and Oracle Solaris* by Darryl Gove (Addison-Wesley, 2010)
- *The Art of Multiprocessor Programming* by Maurice Herlihy & Nir Shavit (Morgan Kaufmann, 2012)
- *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications* by Clay Breshears (O'Reilly, 2009)
- *Concurrent Programming* by Alan Burns & Geoff Davies (Addison-Wesley, 1993)
- *Distributed Algorithms and Protocols* by Michel Raynal (Wiley, 1988)
- *Operating System Concepts* by Silberschatz, Peterson, & Galvin (Addison-Wesley, 1991)
- *Principles of Concurrent Programming* by M. Ben-Ari (Prentice-Hall, 1982)
- *Programming with Threads* by Steve Kleiman, Devang Shah, & Bart Smaalders (Prentice Hall, 1996)

  Programming with POSIX Threads by David R. Butenhof (Addison-Wesley Professional, 1997)

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Covering Multithreading Basics

The word *multithreading* can be translated as *multiple threads of control* or *multiple flows of control*. While a traditional UNIX process contains a single thread of control, multithreading (MT) separates a process into many execution threads. Each of these threads runs independently.

This chapter explains some multithreading terms, benefits, and concepts. If you are ready to start using multithreading, skip to Chapter 2, "Basic Threads Programming."

If you need in-depth information about multithreaded programming, see the "Related Books" on page 15 section of the preface.

- "Multithreading Terms" on page 17
- "Oracle Solaris Multithreading Libraries and Standards" on page 19
- "Benefiting From Multithreading" on page 20
- "Multithreading Concepts" on page 21

Multithreading Terms

Table 1–1 introduces some of the terms that are used in this book.

TABLE 1–1 Multithreading Terms

Term	Definition
Process	The UNIX environment, such as 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.
POSIX `pthreads`	A threads interface that is POSIX threads compliant. See "Oracle Solaris Multithreading Libraries and Standards" on page 19 for more information.

TABLE 1–1 Multithreading Terms *(Continued)*

Term	Definition
Oracle Solaris `threads`	An Oracle Solaris threads interface that is not POSIX threads compliant. A predecessor of pthreads.
Single-threaded	Restricts access to a single thread. Execution is through sequential processing, limited to one thread of control.
Multithreading	Allows access to two or more threads. Execution occurs in more than one thread of control, using parallel or concurrent processing.
User-level or Application-level threads	Threads managed by threads routines in user space, as opposed to kernel space. The POSIX pthreads and Oracle Solaris threads APIs are used to create and handle user threads. In this manual, and in general, a thread is a user-level thread. **Note –** Because this manual is for application programmers, kernel thread programming is not discussed.
Lightweight processes	Kernel threads, also called LWPs, that execute kernel code and system calls. LWPs are managed by the system thread scheduler, and cannot be directly controlled by the application programmer. Beginning with Solaris 9, every user-level thread has a dedicated LWP. This is known as a 1:1 thread model.
Bound thread (obsolete term)	Prior to Solaris 9, a user-level thread that is permanently bound to one LWP. Beginning with Solaris 9, every thread has a dedicated LWP, so all threads are bound threads. The concept of an unbound thread no longer exists.
Unbound thread (obsolete term)	Prior to Solaris 9, a user-level thread that is not necessarily bound to one LWP. Beginning with Solaris 9, every thread has a dedicated LWP, so the concept of unbound threads no longer exists.
Attribute object	Contains opaque data types and related manipulation functions. These data types and functions standardize some of the configurable aspects of POSIX threads, mutual exclusion locks (mutexes), and condition variables.
Mutual exclusion locks	Objects used to lock and unlock access to shared data. Such objects are also known as mutexes.
Condition variables	Objects used to block threads until a change of state.
Read-write locks	Objects used to allow multiple read-only access to shared data, but exclusive access for modification of that data.
Counting semaphore	A memory-based synchronization mechanism in which a non-negative integer count is used to coordinate access by multiple threads to shared resources.

TABLE 1–1 Multithreading Terms *(Continued)*

Term	Definition
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 include time-slicing as a form of virtual parallelism.

Oracle Solaris Multithreading Libraries and Standards

The concept of multithreaded programming goes back to at least the 1960s. Multithreaded programming development on UNIX systems began in the middle 1980s. While agreement existed about what multithreading is and the features necessary to support multithreading, the interfaces used to implement multithreading have varied greatly in the past.

For several years, POSIX (Portable Operating System Interface) 1003.4a worked on standards for multithreaded programming. The standard was eventually ratified and is now part of The Single UNIX Specification (SUS). The latest specification is available at The Open Group website. Beginning with the Oracle Solaris 10 release, the Oracle Solaris OS conforms to The Open Group's UNIX 03 Product Standard, or SUSv3.

Before the POSIX standard was ratified, the Oracle Solaris multithreading API was implemented in the Oracle Solaris `libthread` library, which was developed by Oracle and later became the basis for the UNIX International (UI) threads standard. The `libthread` library was introduced in the Solaris 2.2 release in 1993. Support for the POSIX standard was added with the `libpthread` API in the Solaris 2.5 release in 1995, and both APIs have been available since. The `libthread` and `libpthread` libraries were merged into the standard `libc` C library beginning in the Oracle Solaris 10 release.

The `libthread` and `libpthread` libraries are maintained to provide backward compatibility for both runtime and compilation environments. The `libthread.so.1` and `libpthread.so.1` shared objects are implemented as filters on `libc.so.1`. See the `libthread`(3LIB) and `libpthread`(3LIB) man pages for more information.

While both thread libraries are supported, the POSIX library should be used in most cases. The `threads`(5) man page documents the differences and similarities between POSIX threads and Oracle Solaris threads.

This *Multithreaded Programming Guide* is based on the latest revision of the POSIX standard IEEE Std 1003.1:2001 (also known as ISO/IEC 9945:2003 and as The Single UNIX Specification, Version 3).

Subjects specific to Oracle Solaris threads are covered in the Chapter 6, "Programming With Oracle Solaris Threads."

Benefiting From Multithreading

This section briefly describes the benefits of multithreading.

Multithreading your code can help in the following areas:

- "Improving Application Responsiveness" on page 20
- "Using Multiprocessors Efficiently" on page 20
- "Improving Program Structure" on page 20
- "Using Fewer System Resources" on page 21

Improving Application Responsiveness

Any program in which many activities are not dependent upon each other can be redesigned so that each independent 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 activity.

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 because the operating system takes care of scheduling threads for the number of processors that are available. When multicore processors and multithreaded processors are available, a multithreaded application's performance scales appropriately because the cores and threads are viewed by the OS as processors.

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

Note – In this manual, whenever multiprocessors are discussed, the context applies also to multicore and multithreaded processors unless noted otherwise.

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. For example, a non-threaded program that performs many different tasks might need to devote much of its code just to coordinating the tasks. When the tasks are programmed as threads, the code can be simplified. Multithreaded programs, especially programs that provide service to multiple concurrent users, 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 environment state. Cost of creating and maintaining this large amount of state information 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. This effort includes handling communication between the threads in different processes, or synchronizing their actions. When the threads are in the same process, communication and synchronization becomes much easier.

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 additional 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.

Note – The Message Processing Interface (MPI) might be a more effective approach to achieve multithreading in applications that run across distributed systems. See `http://www-unix.mcs.anl.gov/mpi/` for more information about MPI.

Multithreading Concepts

This section introduces basic concepts of multithreading.

Concurrency and Parallelism

In a multithreaded process on a single processor, the processor can switch execution resources between threads, resulting in concurrent execution. Concurrency indicates that more than one thread is making progress, but the threads are not actually running simultaneously. The switching between threads happens quickly enough that the threads might appear to run simultaneously.

In the same multithreaded process in a shared-memory multiprocessor environment, each thread in the process can run concurrently on a separate processor, resulting in parallel execution, which is true simultaneous execution. When the number of threads in a process is less than or equal to the number of processors available, the operating system's thread support system ensures that each thread runs on a different processor. For example, in a matrix multiplication that is programmed with four threads, and runs on a system that has two dual-core processors, each software thread can run simultaneously on the four processor cores to compute a row of the result at the same time.

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 less expensive than creating a new process because the newly created thread uses the current process address space. The time that is required to switch between threads is less than the time required to switch between processes. A switch between threads is faster because no switching between address spaces occurs.

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

However, this sharing of data leads to a different set of challenges for the programmer. Care must be taken to synchronize threads to protect data from being modified by more than one thread at once, or from being read by some threads while being modified by another thread at the same time. See "Thread Synchronization" on page 24 for more information.

User-Level Threads

Threads are the primary programming interface in multithreaded programming. Threads are visible only from within the process, where the threads share all process resources like address space, open files, and so on.

User-Level Threads State

The following state is unique to each thread.

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

Threads share the process instructions and most of the process data. For that reason, 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, the thread can do so without involving the operating environment.

Note – User-level threads are so 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.

Thread Scheduling

The POSIX standard 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 an execution time quota.

Both `SCHED_FIFO` and `SCHED_RR` are POSIX Realtime extensions. Threads executing with these policies are in the Oracle Solaris Real-Time (RT) scheduling class, normally requiring special privilege. `SCHED_OTHER` is the default scheduling policy. Threads executing with the `SCHED_OTHER` policy are in the traditional Oracle Solaris Time-Sharing (TS) scheduling class.

Oracle Solaris provides other scheduling classes, namely the Interactive timesharing (IA) class, the Fair-Share (FSS) class, and the Fixed-Priority (FX) class. Such specialized classes are not discussed here. See the Oracle Solaris `priocntl(2)` manual page for more information.

See "LWPs and Scheduling Classes" on page 151 for information about the `SCHED_OTHER` policy.

Two scheduling scopes are available: process scope (`PTHREAD_SCOPE_PROCESS`) and system scope (`PTHREAD_SCOPE_SYSTEM`). Threads with differing scope states can coexist on the same system and even in the same process. Process scope causes such threads to contend for resources only with other such threads in the same process. System scope causes such threads to contend with all other threads in the system. In practice, beginning with the Solaris 9 release, the system makes no distinction between these two scopes.

Thread Cancellation

A thread can request the termination of any other thread in the process. The target thread, the one being cancelled, can keep cancellation requests pending as well as perform application-specific cleanup when the thread acts upon the cancellation request.

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.

Thread Synchronization

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

The four synchronization models are mutex locks, read/write 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.
- *Read/write locks* permit concurrent reads and exclusive writes to a protected shared resource. To modify a resource, a thread must first acquire the exclusive write lock. An exclusive write lock is not permitted until all read locks have been released.
- *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 concurrent access to the data protected by the semaphore. When the count is reached, the semaphore causes the calling thread to block until the count changes. A binary semaphore (with a count of one) is similar in operation to a mutex lock.

Using the 64-bit Architecture

For application developers, the major difference between the Oracle Solaris 64-bit and 32–bit environments is the C–language data type model used. The 64-bit data type uses the LP64 model where `longs` and pointers are 64 bits wide. All other fundamental data types remain the same as the data types of the 32–bit implementation. The 32–bit data type uses the ILP32 model where `ints`, `longs`, and pointers are 32 bits.

The following summary briefly describes the major features and considerations for using the 64-bit environment:

- Large Virtual Address Space

  In the 64-bit environment, a process can have up to 64 bits of virtual address space, or 18 exabytes. The larger virtual address space is 4 billion times the current 4 Gbyte maximum of a 32-bit process. Because of hardware restrictions, however, some platforms might not support the full 64 bits of address space.

  A large address space increases the number of threads that can be created with the default stack size. The default stack size is 1 megabyte on 32 bits, 2 megabytes on 64 bits. The number of threads with the default stack size is approximately 2000 threads on a 32–bit system and 8000 billion on a 64-bit system.

- Kernel Memory Readers

  The kernel is an LP64 object that uses 64-bit data structures internally. This means that existing 32-bit applications that use `libkvm`, `/dev/mem`, or `/dev/kmem` do not work properly and must be converted to 64-bit programs.

- `/proc` Restrictions

  A 32-bit program that uses `/proc` is able to look at 32-bit processes but is unable to understand a 64-bit process. The existing interfaces and data structures that describe the process are not large enough to contain the 64-bit quantities. Such programs must be recompiled as 64-bit programs to work for both 32-bit processes and 64-bit processes.
- 64-bit Libraries

  32–bit applications are required to link with 32–bit libraries and 64-bit applications are required to link with 64-bit libraries. With the exception of those libraries that have become obsolete, all of the system libraries are provided in both 32–bit versions and 64-bit versions.
- 64-bit Arithmetic

  64-bit arithmetic has long been available in previous 32–bit Oracle Solaris releases. The 64-bit implementation now provides full 64-bit machine registers for integer operations and parameter passing.
- Large Files

  If an application requires only large file support, the application can remain 32-bit and use the Large Files interface. To take full advantage of 64-bit capabilities, the application must be converted to 64-bit.

CHAPTER 2

Basic Threads Programming

This chapter introduces the basic threads programming routines for POSIX threads. This chapter describes *default threads*, or threads with default attribute values, which are the kind of threads that are most often used in multithreaded programming. This chapter explains how to create and use threads with nondefault attributes.

- "Lifecycle of a Thread" on page 27
- "The Pthreads Library" on page 28

Lifecycle of a Thread

When a thread is created, a new thread of control is added to the current process. Every process has at least one thread of control, in the program's `main()` routine. Each thread in the process runs simultaneously, and has access to the calling process's global data. In addition each thread has its own private attributes and call stack.

To create a new thread, a running thread calls the `pthread_create()` function, and passes a pointer to a function for the new thread to run. One argument for the new thread's function can also be passed, along with thread attributes. The execution of a thread begins with the successful return from the `pthread_create()` function. The thread ends when the function that was called with the thread completes normally.

A thread can also be terminated if the thread calls a `pthread_exit()` routine, or if any other thread calls `pthread_cancel()` to explicitly terminate that thread. A thread can also be terminated by the exit of the process that called the thread.

The Pthreads Library

The Pthreads API library consists of more than 100 functions. See the `pthreads(5)` man page for a full list of the functions, grouped by their usage categories.

This section contains brief descriptions of the functions used for basic threads programming, organized according to the task they perform, and includes links to the man pages of the associated API functions. The following list directs you to the discussion of a particular task.

- "Creating a Default Thread" on page 28
- "Waiting for Thread Termination" on page 30
- "Simple Threads Example" on page 31
- "Detaching a Thread" on page 32
- "Creating a Key for Thread-Specific Data" on page 33
- "Deleting the Thread-Specific Data Key" on page 34
- "Setting Thread-Specific Data" on page 35
- "Getting Thread-Specific Data" on page 36
- "Global and Private Thread-Specific Data Example" on page 36
- "Getting the Thread Identifier" on page 38
- "Comparing Thread IDs" on page 39
- "Calling an Initialization Routine for a Thread" on page 39
- "Yielding Thread Execution" on page 40
- "Setting the Thread Policy and Scheduling Parameters" on page 40
- "Getting the Thread Policy and Scheduling Parameters" on page 41
- "Setting the Thread Priority" on page 42
- "Sending a Signal to a Thread" on page 43
- "Accessing the Signal Mask of the Calling Thread" on page 43
- "Forking Safely" on page 44
- "Terminating a Thread" on page 45
- "Finishing Up" on page 45
- "Cancel a Thread" on page 46
- "Cancelling a Thread" on page 47
- "Enabling or Disabling Cancellation" on page 48
- "Setting Cancellation Type" on page 48
- "Creating a Cancellation Point" on page 49
- "Pushing a Handler Onto the Stack" on page 49
- "Pulling a Handler Off the Stack" on page 50

Creating a Default Thread

When an attribute object is not specified, the object is `NULL`, and the default thread is created with the following attributes:

- Process scope

- Nondetached
- A default stack and stack size
- A priority of zero

You can also create a default attribute object with `pthread_attr_init()` , and then use this attribute object to create a default thread. See the section "Initializing Attributes" on page 52 for details.

pthread_create Syntax

Use `pthread_create(3C)` to add a new thread of control to the current process.

```
int pthread_create(pthread_t *restrict tid, const pthread_attr_t
    *restrict tattr, void*(*start_routine)(void *), void *restrict arg);
```

```
#include <pthread.h>

pthread_attr_t() tattr;
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;

/* default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);
/* default behavior specified*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

The `pthread_create()` function is called with *attr* that has the necessary state behavior. *start_routine* is the function with which the new thread begins execution. When *start_routine* returns, the thread exits with the exit status set to the value returned by *start_routine*. See "`pthread_create` Syntax" on page 29.

When `pthread_create()` is successful, the ID of the created thread is stored in the location referred to as *tid*.

When you call `pthread_create()` with either a `NULL` attribute argument or a default attribute, `pthread_create()` creates a default thread. When *tattr* is initialized, the thread acquires the default behavior.

pthread_create Return Values

`pthread_create()` returns zero when the call completes successfully. Any other return value indicates that an error occurred. When any of the following conditions are detected, `pthread_create()` fails and returns the corresponding value.

`EAGAIN`

Description: A system limit is exceeded, such as when too many threads have been created.

`EINVAL`

Description: The value of *tattr* is invalid.

`EPERM`

Description: The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

Waiting for Thread Termination

The `pthread_join()` function blocks the calling thread until the specified thread terminates.

pthread_join Syntax

Use `pthread_join`(3C) to wait for a thread to terminate.

```
int pthread_join(pthread_t tid, void **status);
```

```
#include <pthread.h>

pthread_t tid;
int ret;
void *status;

/* waiting to join thread "tid" with status */
ret = pthread_join(tid, &status);

/* waiting to join thread "tid" without status */
ret = pthread_join(tid, NULL);
```

The specified thread must be in the current process and must not be detached. For information on thread detachment, see "Setting Detach State" on page 54.

When *status* is not `NULL`, *status* points to a location that is set to the exit status of the terminated thread when `pthread_join()` returns successfully.

If multiple threads wait for the same thread to terminate, all the threads wait until the target thread terminates. Then one waiting thread returns successfully. The other waiting threads fail with an error of `ESRCH`.

After `pthread_join()` returns, any data storage associated with the terminated thread can be reclaimed by the application.

pthread_join Return Values

`pthread_join()` returns zero when the call completes successfully. Any other return value indicates that an error occurred. When any of the following conditions are detected, `pthread_join()` fails and returns the corresponding value.

`ESRCH`

Description: No thread could be found corresponding to the given thread ID.

`EDEADLK`

Description: A deadlock would exist, such as a thread waits for itself or thread A waits for thread B and thread B waits for thread A.

`EINVAL`

Description: The thread corresponding to the given thread ID is a detached thread.

`pthread_join()` works only for target threads that are nondetached. When no reason exists to synchronize with the termination of a particular thread, then that thread should be detached.

Simple Threads Example

In Example 2–1, one thread executes the procedure at the top, creating a helper thread that executes the procedure `fetch()`. The `fetch()` procedure executes a complicated database lookup and might take some time.

The main thread awaits the results of the lookup but has other work to do in the meantime. So, the main thread perform those other activities and then waits for its helper to complete its job by executing `pthread_join()`.

An argument, *pbe*, to the new thread is passed as a stack parameter. The thread argument can be passed as a stack parameter because the main thread waits for the spun-off thread to terminate. However, the preferred method is to use `malloc` to allocate storage from the heap instead of passing an address to thread stack storage. If the argument is passed as an address to thread stack storage, this address might be invalid or be reassigned if the thread terminates.

EXAMPLE 2–1 Simple Threads Program

```
void mainline (...)
{
        struct phonebookentry *pbe;
        pthread_attr_t tattr;
        pthread_t helper;
        void *status;

        pthread_create(&helper, NULL, fetch, &pbe);

            /* do something else for a while */
```

EXAMPLE 2–1 Simple Threads Program *(Continued)*

```
        pthread_join(helper, &status);
        /* it's now safe to use result */
}

void *fetch(struct phonebookentry *arg)
{
        struct phonebookentry *npbe;
        /* fetch value from a database */

        npbe = search (prog_name)
            if (npbe != NULL)
                *arg = *npbe;
        pthread_exit(0);
}

struct phonebookentry {
        char name[64];
        char phonenumber[32];
        char flags[16];
}
```

Detaching a Thread

`pthread_detach(3C)` is an alternative to `pthread_join(3C)` to reclaim storage for a thread that is created with a *detachstate* attribute set to `PTHREAD_CREATE_JOINABLE`.

pthread_detach Syntax

```
int pthread_detach(pthread_t tid);
```

```
#include <pthread.h>

pthread_t tid;
int ret;

/* detach thread tid */
ret = pthread_detach(tid);
```

The `pthread_detach()` function is used to indicate to your application that storage for the thread *tid* can be reclaimed when the thread terminates. Threads should be detached when they are no longer needed. If *tid* has not terminated, `pthread_detach()` does not cause the thread to terminate.

pthread_detach Return Values

`pthread_detach()` returns zero when the call completes successfully. Any other return value indicates that an error occurred. When any of the following conditions is detected, `pthread_detach()` fails and returns the corresponding value.

EINVAL

Description: *tid* is a detached thread.

ESRCH

Description: *tid* is not a valid, undetached thread in the current process.

Creating a Key for Thread-Specific Data

Single-threaded C programs have two basic classes of data: local data and global data. For multithreaded C programs a third class, *thread-specific data*, is added. Thread-specific data is very much like global data, except that the data is private to a thread.

Note – The Oracle Solaris OS supports an alternative facility that allows a thread to have a private copy of a global variable. This mechanism is referred to as *thread local storage* (TLS). The keyword `__thread` is used to declare variables to be thread-local, and the compiler automatically arranges for these variables to be allocated on a per-thread basis. See Chapter 14, "Thread-Local Storage," in *Linker and Libraries Guide* for more information.

Thread-specific data (TSD) is maintained on a per-thread basis. TSD is the only way to define and refer to data that is private to a thread. Each thread-specific data item is associated with a *key* that is global to all threads in the process. By using the *key*, a thread can access a pointer (*void* *) maintained per-thread.

pthread_key_create Syntax

```
int pthread_key_create(pthread_key_t *key,
    void (*destructor) (void *));

#include <pthread.h>

pthread_key_t  key;
int ret;

/* key create without destructor */
ret = pthread_key_create(&key, NULL);

/* key create with destructor */
ret = pthread_key_create(&key, destructor);
```

Use `pthread_key_create(3C)` to allocate a *key* that is used to identify thread-specific data in a process. The key is global to all threads in the process. When the thread-specific data is created, all threads initially have the value `NULL` associated with the key.

Call `pthread_key_create()` once for each key before using the key. No implicit synchronization exists for the keys shared by all threads in a process.

Once a key has been created, each thread can bind a value to the key. The values are specific to the threads and are maintained for each thread independently. The per-thread binding is deallocated when a thread terminates if the key was created with a `destructor` function.

When `pthread_key_create()` returns successfully, the allocated key is stored in the location pointed to by *key*. The caller must ensure that the storage and access to this key are properly synchronized.

An optional destructor function, `destructor`, can be used to free stale storage. If a key has a non-`NULL` `destructor` function and the thread has a non-`NULL` value associated with that key, the `destructor` function is called with the current associated value when the thread exits. The order in which the `destructor` functions are called is unspecified.

pthread_key_create Return Values

`pthread_key_create()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occur, `pthread_key_create()` fails and returns the corresponding value.

`EAGAIN`

Description: The *key* name space is exhausted.

`ENOMEM`

Description: Insufficient virtual memory is available in this process to create a new key.

Deleting the Thread-Specific Data Key

Use `pthread_key_delete(3C)` to destroy an existing thread-specific data key. Any memory associated with the key can be freed because the key has been invalidated. Reference to an invalid key returns an error.

pthread_key_delete Syntax

```
int pthread_key_delete(pthread_key_t key);

#include <pthread.h>

pthread_key_t key;
int ret;

/* key previously created */
ret = pthread_key_delete(key);
```

If a *key* has been deleted, any reference to the key with the `pthread_setspecific()` or `pthread_getspecific()` call yields undefined results.

The programmer must free any thread-specific resources before calling the `pthread_key_delete()` function. This function does not invoke any of the destructors. Repeated calls to `pthread_key_create()` and `pthread_key_delete()` can cause a problem.

The problem occurs because, in the Oracle Solaris implementation, a *key* value is never reused after `pthread_key_delete()` marks it as invalid. Every `pthread_key_create()` allocates a new key value and allocates more internal memory to hold the key information. An infinite loop of `pthread_key_create()` ... `pthread_key_delete()` will eventually exhaust all memory. If possible, call `pthread_key_create()` only once for each desired key and never call `pthread_key_delete()`.

pthread_key_delete Return Values

`pthread_key_delete()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, `pthread_key_delete()` fails and returns the corresponding value.

`EINVAL`

Description: The *key* value is invalid.

Setting Thread-Specific Data

Use `pthread_setspecific(3C)` to set the thread-specific binding to the specified thread-specific data key.

pthread_setspecific Syntax

```
int pthread_setspecific(pthread_key_t key, const void *value);
```

```
#include <pthread.h>

pthread_key_t key;
void *value;
int ret;

/* key previously created */
ret = pthread_setspecific(key, value);
```

pthread_setspecific Return Values

`pthread_setspecific()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occur, `pthread_setspecific()` fails and returns the corresponding value.

`ENOMEM`

Description: Insufficient virtual memory is available.

`EINVAL`

Description: *key* is invalid.

Note – `pthread_setspecific()` does not free its storage when a new binding is set. The existing binding must be freed, otherwise a memory leak can occur.

Getting Thread-Specific Data

Use `pthread_getspecific(3C)` to get the calling thread's binding for *key*, and store the binding in the location pointed to by *value*.

pthread_getspecific Syntax

```
void *pthread_getspecific(pthread_key_t key);
```

```
#include <pthread.h>

pthread_key_t key;
void *value;

/* key previously created */
value = pthread_getspecific(key);
```

pthread_getspecific Return Values

`pthread_getspecific` returns no errors.

Global and Private Thread-Specific Data Example

Example 2–2 shows an excerpt from a multithreaded program. This code is executed by any number of threads, but the code has references to two global variables, *errno* and *mywindow*. These global values really should be references to items private to each thread.

EXAMPLE 2–2 Thread-Specific Data–Global but Private

```
body() {
    ...

    while (write(fd, buffer, size) == -1) {
        if (errno != EINTR) {
            fprintf(mywindow, "%s\n", strerror(errno));
            exit(1);
        }
    }

    ...
```

EXAMPLE 2–2 Thread-Specific Data–Global but Private (*Continued*)

```
}
```

References to `errno` should get the system error code from the routine called by this thread, not by some other thread. Including the header file `errno.h` causes a reference to `errno` to be a reference to a thread-private instance of `errno`, so that references to `errno` by one thread refer to a different storage location than references to `errno` by other threads.

The *mywindow* variable refers to a `stdio` stream that is connected to a window that is private to the referring thread. So, as with `errno`, references to *mywindow* by one thread should refer to a different storage location than references to *mywindow* by other threads. Ultimately, the reference is to a different window. The only difference here is that the system takes care of `errno` , but the programmer must handle references for *mywindow* .

The next example shows how the references to *mywindow* work. The preprocessor converts references to *mywindow* into invocations of the `_mywindow()` procedure.

This routine in turn invokes `pthread_getspecific()`. `pthread_getspecific()` receives the *mywindow_key* global variable and *win* an output parameter that receives the identity of this thread's window.

EXAMPLE 2–3 Turning Global References Into Private References

```
thread_key_t mywin_key;

FILE *_mywindow(void) {
 FILE *win;
 win = pthread_getspecific(mywin_key);
 return(win);
 }
#define mywindow _mywindow()

void routine_uses_win( FILE *win) {
 ...
}
void thread_start(...) {
 ...
 make_mywin();
 ...
 routine_uses_win( mywindow )
 ...
}
```

The *mywin_key* variable identifies a class of variables for which each thread has its own private copy. These variables are thread-specific data. Each thread calls `make_mywin()` to initialize its window and to arrange for its instance of *mywindow* to refer to the thread-specific data.

Once this routine is called, the thread can safely refer to *mywindow* and, after `_mywindow()`, the thread gets the reference to its private window. References to *mywindow* behave as if direct references were made to data private to the thread.

Example 2–4 shows how to set up the reference.

EXAMPLE 2–4 Initializing the Thread-Specific Data

```
void make_mywindow(void) {
    FILE **win;
    static pthread_once_t mykeycreated = PTHREAD_ONCE_INIT;

    pthread_once(&mykeycreated, mykeycreate);

    win = malloc(sizeof(*win));
    create_window(win, ...);

    pthread_setspecific(mywindow_key, win);
}

void mykeycreate(void) {
    pthread_key_create(&mywindow_key, free_key);
}

void free_key(void *win) {
    free(win);
}
```

First, get a unique value for the key, *mywin_key*. This key is used to identify the thread-specific class of data. The first thread to call `make_mywin()` eventually calls `pthread_key_create()`, which assigns to its first argument a unique *key*. The second argument is a `destructor` function that is used to deallocate a thread's instance of this thread-specific data item once the thread terminates.

The next step is to allocate the storage for the caller's instance of this thread-specific data item. Having allocated the storage, calling `create_window()` sets up a window for the thread. *win* points to the storage allocated for the window. Finally, a call is made to `pthread_setspecific()`, which associates *win* with the key.

Subsequently, whenever the thread calls `pthread_getspecific()` to pass the global *key*, the thread gets the value that is associated with this key by this thread in an earlier call to `pthread_setspecific()`.

When a thread terminates, calls are made to the `destructor` functions that were set up in `pthread_key_create()`. Each `destructor` function is called only if the terminating thread established a value for the *key* by calling `pthread_setspecific()`.

Getting the Thread Identifier

Use `pthread_self`(3C) to get the thread identifier of the calling thread.

pthread_self Syntax

```
pthread_t  pthread_self(void);
```

```
#include <pthread.h>

pthread_t tid;

tid = pthread_self();
```

pthread_self Return Values

`pthread_self()` returns the thread identifier of the calling thread.

Comparing Thread IDs

Use `pthread_equal`(3C) to compare the thread identification numbers of two threads.

pthread_equal Syntax

```
int  pthread_equal(pthread_t tid1, pthread_t tid2);
```

```
#include <pthread.h>

pthread_t tid1, tid2;
int ret;

ret = pthread_equal(tid1, tid2);
```

pthread_equal Return Values

`pthread_equal()` returns a nonzero value when *tid1* and *tid2* are equal, otherwise, `0` is returned. When either *tid1* or *tid2* is an invalid thread identification number, the result is unpredictable.

Calling an Initialization Routine for a Thread

Use `pthread_once`(3C) in a threaded process to call an initialization routine the first time `pthread_once` is called. Subsequent calls to `pthread_once()` from any thread in the process have no effect.

pthread_once Syntax

```
int  pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;
int ret;

ret = pthread_once(&once_control,
init_routine);
```

The *once_control* parameter determines whether the associated initialization routine has been called.

pthread_once Return Values

`pthread_once()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, `pthread_once()` fails and returns the corresponding value.

`EINVAL`

Description: *once_control* or *init_routine* is `NULL`.

Yielding Thread Execution

Use `sched_yield` to cause the current thread to yield its execution in favor of another thread with the same or greater priority. If no such threads are ready to run, the calling thread continues to run. The `sched_yield()` function is not part of the Pthread API, but is a function in the Realtime Library Functions. You must include `<sched.h>` to use `sched_yield()`.

sched_yield Syntax

```
int  sched_yield(void);
```

```
#include <sched.h>
int ret;
ret = sched_yield();
```

sched_yield Return Values

`sched_yield()` returns zero after completing successfully. Otherwise, -1 is returned and *errno* is set to indicate the error condition.

Setting the Thread Policy and Scheduling Parameters

Use `pthread_setschedparam(3C)` to modify the scheduling policy and scheduling parameters of an individual thread.

pthread_setschedparam Syntax

```
int pthread_setschedparam(pthread_t tid, int  policy,
    const struct sched_param *param);
```

```
#include <pthread.h>

pthread_t tid;
```

```
int ret;
struct sched_param param;
int priority;

/* sched_priority will be the priority of the thread */
sched_param.sched_priority = priority;
policy = SCHED_OTHER;

/* scheduling parameters of target thread */
ret = pthread_setschedparam(tid,
policy, &param);
```

Supported policies are `SCHED_FIFO`, `SCHED_RR`, and `SCHED_OTHER`.

pthread_setschedparam Return Values

`pthread_setschedparam()` returns zero after completing successfully. Any other return value indicates that an error occurred. When either of the following conditions occurs, the `pthread_setschedparam()` function fails and returns the corresponding value.

`EINVAL`

Description: The value of the attribute being set is not valid.

`EPERM`

Description: The caller does not have the appropriate permission to set either the scheduling parameters or the scheduling policy of the specified thread.

`ESRCH`

Description: The value specified by *tid* does not refer to an existing thread.

Getting the Thread Policy and Scheduling Parameters

`pthread_getschedparam`(3C) gets the scheduling policy and scheduling parameters of an individual thread.

pthread_getschedparam Syntax

```
int  pthread_getschedparam(pthread_t tid, int *restrict policy,
    struct sched_param *restrict param);

#include <pthread.h>

pthread_t tid;
sched_param param;
int priority;
int policy;
int ret;

/* scheduling parameters of target thread */
```

```
ret = pthread_getschedparam (tid, &policy, &param);

/* sched_priority contains the priority of the thread */
priority = param.sched_priority;
```

pthread_getschedparam Return Values

`pthread_getschedparam()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`ESRCH`

Description: The value specified by *tid* does not refer to an existing thread.

Setting the Thread Priority

`pthread_setschedprio`(3C) sets the scheduling priority for the specified thread.

pthread_setschedprio Syntax

```
int pthread_setschedprio(pthread_t tid, int prio);

#include <pthread.h>

pthread_t tid;
int prio;
int ret;
```

pthread_setschedprio Return Values

`pthread_setschedprio()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value of *prio* is invalid for the scheduling policy of the specified thread.

`ENOTSUP`

Description: An attempt was made to set the priority to an unsupported value.

`EPERM`

Description: The caller does not have the appropriate permission to set the scheduling priority of the specified thread.

`ESRCH`

Description: The value specified by *tid* does not refer to an existing thread.

Sending a Signal to a Thread

Use `pthread_kill(3C)` to send a signal to a thread.

pthread_kill Syntax

```
int  pthread_kill(pthread_t tid, int sig);

#include <pthread.h>
#include <signal.h>

int sig;
pthread_t tid;
int ret;

ret = pthread_kill(tid,
sig);
```

`pthread_kill()` sends the signal *sig* to the thread specified by *tid*. *tid* must be a thread within the same process as the calling thread. The*sig* argument must be from the list that is given in `signal.h(3HEAD)`.

When *sig* is zero, error checking is performed but no signal is actually sent. This error checking can be used to check the validity of *tid*.

pthread_kill Return Values

`pthread_kill()` returns zero after completing successfully. Any other return value indicates that an error occurred. When either of the following conditions occurs, `pthread_kill()` fails and returns the corresponding value.

`EINVAL`

Description: *sig* is not a valid signal number.

`ESRCH`

Description: *tid* cannot be found in the current process.

Accessing the Signal Mask of the Calling Thread

Use `pthread_sigmask(3C)` to change or examine the signal mask of the calling thread.

pthread_sigmask Syntax

```
int pthread_sigmask(int how, const sigset_t *new, sigset_t *old);

#include <pthread.h>
#include <signal.h>
```

```
int ret;
sigset_t old, new;

ret = pthread_sigmask(SIG_SETMASK, &new, &old); /* set new mask */
ret = pthread_sigmask(SIG_BLOCK, &new, &old); /* blocking mask */
ret = pthread_sigmask(SIG_UNBLOCK, &new, &old); /* unblocking */
```

how determines how the signal set is changed. *how* can have one of the following values:

- `SIG_BLOCK`. Add *new* to the current signal mask, where *new* indicates the set of signals to block.
- `SIG_UNBLOCK`. Delete *new* from the current signal mask, where *new* indicates the set of signals to unblock.
- `SIG_SETMASK` . Replace the current signal mask with *new*, where *new* indicates the new signal mask.

When the value of *new* is `NULL`, the value of *how* is not significant. The signal mask of the thread is unchanged. To inquire about currently blocked signals, assign a `NULL` value to the *new* argument.

The *old* variable points to the space where the previous signal mask is stored, unless *old* is `NULL`.

pthread_sigmask Return Values

`pthread_sigmask()` returns zero when the call completes successfully. Any other return value indicates that an error occurred. When the following condition occurs, `pthread_sigmask()` fails and returns the corresponding value.

`EINVAL`

Description: The value of *how* is not defined and *old* is `NULL`.

Forking Safely

See the discussion about `pthread_atfork`(3C) in "Solution: pthread_atfork" on page 148.

pthread_atfork Syntax

```
int pthread_atfork(void (*prepare) (void), void (*parent) (void),
    void (*child) (void) );
```

pthread_atfork Return Values

`pthread_atfork()` returns zero when the call completes successfully. Any other return value indicates that an error occurred. When the following condition occurs, `pthread_atfork()` fails and returns the corresponding value.

`ENOMEM`

Description: Insufficient table space exists to record the fork handler addresses.

Terminating a Thread

Use pthread_exit(3C) to terminate a thread.

pthread_exit Syntax

```
void     pthread_exit(void *status);
```

```
#include <pthread.h>
void *status;
pthread_exit(status); /* exit with status */
```

The pthread_exit() function terminates the calling thread. All thread-specific data bindings are released. If the calling thread is not detached, then the thread's ID and the exit status specified by *status* are retained until your application calls pthread_join() to wait for the thread. Otherwise, *status* is ignored. The thread's ID can be reclaimed immediately. For information on thread detachment, see "Setting Detach State" on page 54.

pthread_exit Return Values

The calling thread terminates with its exit status set to the contents of *status*.

Finishing Up

A thread can terminate its execution in the following ways:

- By returning from its first (outermost) procedure, the threads start routine. See pthread_create.
- By calling pthread_exit(), supplying an exit status.
- By termination with POSIX cancel functions. See pthread_cancel().

The default behavior of a thread is to linger until some other thread has acknowledged its demise by "joining" with the lingering thread. This behavior is the same as the default pthread_create() attribute that is nondetached, see pthread_detach. The result of the join is that the joining thread picks up the exit status of the terminated thread and the terminated thread vanishes.

An important special case arises when the initial thread, calling main(), returns from calling main() or calls exit(). This action causes the entire process to be terminated, along with all its threads. So, take care to ensure that the initial thread does not return from main() prematurely.

Note that when the main thread merely calls pthread_exit, the main thread terminates itself only. The other threads in the process, as well as the process, continue to exist. The process terminates when all threads terminate.

Cancel a Thread

Cancellation allows a thread to request the termination of any other thread in the process. Cancellation is an option when all further operations of a related set of threads are undesirable or unnecessary.

One example of thread cancellation is an asynchronously generated cancel condition, such as, when a user requesting to close or exit a running application. Another example is the completion of a task undertaken by a number of threads. One of the threads might ultimately complete the task while the others continue to operate. Since the running threads serve no purpose at that point, these threads should be cancelled.

Cancellation Points

Be careful to cancel a thread only when cancellation is safe. The pthreads standard specifies several cancellation points, including:

- Programmatically, establish a thread cancellation point through a `pthread_testcancel` call.
- Threads waiting for the occurrence of a particular condition in `pthread_cond_wait` or `pthread_cond_timedwait(3C)`.
- Threads blocked on `sigwait(2)`.
- Some standard library calls. In general, these calls include functions in which threads can block. See the `cancellation(5)` man page for a list.

Cancellation is enabled by default. At times, you might want an application to disable cancellation. Disabled cancellation has the result of deferring all cancellation requests until cancellation requests are enabled again.

See “`pthread_setcancelstate` Syntax” on page 48 for information about disabling cancellation.

Placing Cancellation Points

Dangers exist in performing cancellations. Most deal with properly restoring invariants and freeing shared resources. A thread that is cancelled without care might leave a mutex in a locked state, leading to a deadlock. Or a cancelled thread might leave a region of allocated memory with no way to identify the memory and therefore unable to free the memory.

The standard C library specifies a cancellation interface that permits or forbids cancellation programmatically. The library defines *cancellation points* that are the set of points at which cancellation can occur. The library also allows the scope of *cancellation handlers* to be defined so that the handlers are sure to operate when and where intended. The cancellation handlers provide clean up services to restore resources and state to a condition that is consistent with the point of origin.

Placement of cancellation points and the effects of cancellation handlers must be based on an understanding of the application. A mutex is explicitly not a cancellation point and should be held only for the minimal essential time.

Limit regions of asynchronous cancellation to sequences with no external dependencies that could result in dangling resources or unresolved state conditions. Take care to restore cancellation state when returning from some alternate, nested cancellation state. The interface provides features to facilitate restoration: `pthread_setcancelstate(3C)` preserves the current cancel state in a referenced variable, `pthread_setcanceltype(3C)` preserves the current cancel type in the same way.

Cancellations can occur under three different circumstances:

- Asynchronously
- At various points in the execution sequence as defined by the standard
- At a call to `pthread_testcancel()`

By default, cancellation can occur only at well-defined points as defined by the POSIX standard.

In all cases, take care that resources and state are restored to a condition consistent with the point of origin.

Cancelling a Thread

Use `pthread_cancel`(3C) to cancel a thread.

pthread_cancel Syntax

```
int pthread_cancel(pthread_t thread);
```

```
#include <pthread.h>

pthread_t thread;
int ret;

ret = pthread_cancel(thread);
```

How the cancellation request is treated depends on the state of the target thread. Two functions, `pthread_setcancelstate(3C)` and `pthread_setcanceltype(3C)`, determine that state.

pthread_cancel Return Values

`pthread_cancel()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`ESRCH`
: **Description:** No thread could be found corresponding to that specified by the given thread ID.

Enabling or Disabling Cancellation

Use pthread_setcancelstate(3C) to enable or disable thread cancellation. When a thread is created, thread cancellation is enabled by default.

pthread_setcancelstate Syntax

```
int pthread_setcancelstate(int state, int *oldstate);
```

```
#include <pthread.h>

int oldstate;
int ret;

/* enabled */
ret = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);

/* disabled */
ret = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &oldstate);
```

pthread_setcancelstate Return Values

`pthread_setcancelstate()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the `pthread_setcancelstate()` function fails and returns the corresponding value.

`EINVAL`

Description: The state is not `PTHREAD_CANCEL_ENABLE` or `PTHREAD_CANCEL_DISABLE`.

Setting Cancellation Type

Use pthread_setcanceltype(3C) to set the cancellation type to either deferred or asynchronous mode.

pthread_setcanceltype Syntax

```
int pthread_setcanceltype(int type, int *oldtype);
```

```
#include <pthread.h>

int oldtype;
int ret;

/* deferred mode */
ret = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);

/* async mode*/
ret = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, &oldtype);
```

When a thread is created, the cancellation type is set to deferred mode by default. In deferred mode, the thread can be cancelled only at cancellation points. In asynchronous mode, a thread can be cancelled at any point during its execution. The use of asynchronous mode is discouraged.

pthread_setcanceltype Return Values

`pthread_setcanceltype()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The type is not `PTHREAD_CANCEL_DEFERRED` or `PTHREAD_CANCEL_ASYNCHRONOUS`.

Creating a Cancellation Point

Use `pthread_testcancel(3C)` to establish a cancellation point for a thread.

pthread_testcancel Syntax

```
void pthread_testcancel(void);
```

```
#include <pthread.h>
```

```
pthread_testcancel();
```

The `pthread_testcancel()` function is effective when thread cancellation is enabled and in deferred mode. `pthread_testcancel()` has no effect if called while cancellation is disabled.

Be careful to insert `pthread_testcancel()` only in sequences where thread cancellation is safe. In addition to programmatically establishing cancellation points through the `pthread_testcancel()` call, the pthreads standard specifies several cancellation points. See "Cancellation Points" on page 46 for more details.

pthread_testcancel Return Values

`pthread_testcancel()` has no return value.

Pushing a Handler Onto the Stack

Use cleanup handlers to restore conditions to a state that is consistent with that state at the point of origin. This consistent state includes cleaning up allocated resources and restoring invariants. Use the `pthread_cleanup_push(3C)` and `pthread_cleanup_pop(3C)` functions to manage the handlers.

Cleanup handlers are pushed and popped in the same lexical scope of a program. The push and pop should always match. Otherwise, compiler errors are generated.

pthread_cleanup_push Syntax

Use `pthread_cleanup_push`(3C) to push a cleanup handler onto a cleanup stack (LIFO).

```
void pthread_cleanup_push(void(*routine)(void *), void *args);

#include <pthread.h>

/* push the handler "routine" on cleanup stack */
pthread_cleanup_push (routine, arg);
```

pthread_cleanup_push Return Values

`pthread_cleanup_push()` has no return value.

Pulling a Handler Off the Stack

Use `pthread_cleanup_pop`(3C) to pull the cleanup handler off the cleanup stack.

pthread_cleanup_pop Syntax

```
void pthread_cleanup_pop(int execute);

#include <pthread.h>

/* pop the "func" out of cleanup stack and execute "func" */
pthread_cleanup_pop (1);

/* pop the "func" and DONT execute "func" */
pthread_cleanup_pop (0);
```

A nonzero argument in the pop function removes the handler from the stack and executes the handler. An argument of zero pops the handler without executing the handler.

`pthread_cleanup_pop()` is effectively called with a nonzero argument when a thread either explicitly or implicitly calls `pthread_exit()` or when the thread accepts a cancel request.

pthread_cleanup_pop Return Values

`pthread_cleanup_pop()` has no return values.

CHAPTER 3

Thread Attributes

The previous chapter covered the basics of threads creation using default attributes. This chapter discusses setting attributes at thread creation time.

Attribute Object

Attributes provide a way to specify behavior that is different from the default thread creation behavior. When a thread is created with `pthread_create(3C)` or when a synchronization variable is initialized, an attribute object can be specified. The defaults are usually sufficient.

An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type.

Once an attribute is initialized and configured, the attribute has process-wide scope. The suggested method for using attributes is to configure all required state specifications at one time in the early stages of program execution. The appropriate attribute object can then be referred to as needed.

The use of attribute objects provides two primary advantages.

- Using attribute objects adds to code portability.

  Even though supported attributes might vary between implementations, you need not modify function calls that create thread entities. These function calls do not require modification because the attribute object is hidden from the interface.

  If the target port supports attributes that are not found in the current port, provision must be made to manage the new attributes. Management of these attributes is an easy porting task because attribute objects need only be initialized once in a well-defined location.
- State specification in an application is simplified.

  As an example, consider that several sets of threads might exist within a process. Each set of threads provides a separate service. Each set has its own state requirements.

At some point in the early stages of the application, a thread attribute object can be initialized for each set. All future thread creations will then refer to the attribute object that is initialized for that type of thread. The initialization phase is simple and localized. Any future modifications can be made quickly and reliably.

Attribute objects require attention at process exit time. When the object is initialized, memory is allocated for the object. This memory must be returned to the system. The `pthreads` standard provides function calls to destroy attribute objects.

Pthreads functions can be used to manipulate thread attribute objects. The functions are described in the following sections.

- "Initializing Attributes" on page 52
- "Destroying Attributes" on page 54
- "Setting Detach State" on page 54
- "Getting the Detach State" on page 55
- "Setting the Stack Guard Size" on page 56
- "Getting the Stack Guard Size" on page 57
- "Setting the Scope" on page 57
- "Getting the Scope" on page 58
- "Setting the Thread Concurrency Level" on page 59
- "Getting the Thread Concurrency Level" on page 59
- "Setting the Scheduling Policy" on page 60
- "Getting the Scheduling Policy" on page 61
- "Setting the Inherited Scheduling Policy" on page 62
- "Getting the Inherited Scheduling Policy" on page 62
- "Setting the Scheduling Parameters" on page 63
- "Getting the Scheduling Parameters" on page 64
- "About Stacks" on page 65
- "Setting the Stack Size" on page 66
- "Getting the Stack Size" on page 67
- "Setting the Stack Address and Size" on page 68
- "Getting the Stack Address and Size" on page 69

Initializing Attributes

Use `pthread_attr_init(3C)` to initialize object attributes to their default values. The storage is allocated by the thread system during execution.

pthread_attr_init Syntax

```
int pthread_attr_init(pthread_attr_t *tattr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
```

```
int ret;

/* initialize an attribute to the default value */
ret = pthread_attr_init(&tattr);
```

Table 3–1 shows the default values for attributes (*tattr*).

TABLE 3–1 Default Attribute Values for *tattr*

Attribute	Value	Result
scope	`PTHREAD_SCOPE_PROCESS`	New thread contends with other threads in the process.
detachstate	`PTHREAD_CREATE_JOINABLE`	Completion status and thread *ID* are preserved after the thread exits.
stackaddr	`NULL`	New thread has system-allocated stack address.
stacksize	0	New thread has system-defined stack size.
priority	0	New thread has priority 0.
inheritsched	`PTHREAD_EXPLICIT_SCHED`	New thread does not inherit parent thread scheduling priority.
schedpolicy	`SCHED_OTHER`	New thread uses the traditional Oracle Solaris time-sharing (TS) scheduling class.
guardsize	`PAGESIZE`	Stack overflow protection.

Note – The default value for the *inheritsched* attribute might change from `PTHREAD_EXPLICIT_SCHED` to `PTHREAD_INHERIT_SCHED` in a future Oracle Solaris release. You should call `pthread_attr_setinheritsched()` to set the *inheritsched* attribute to the value you want rather than accepting the default, in order to avoid any potential problems caused by this change.

pthread_attr_init Return Values

`pthread_attr_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

ENOMEM

Description: Returned when not enough memory is allocated to initialize the thread attributes object.

Destroying Attributes

Use pthread_attr_destroy(3C) to remove the storage that was allocated during initialization. The attribute object becomes invalid.

pthread_attr_destroy Syntax

```
int pthread_attr_destroy(pthread_attr_t *tattr);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* destroy an attribute */
ret = pthread_attr_destroy(&tattr);
```

pthread_attr_destroy Return Values

pthread_attr_destroy() returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL

Description: Indicates that the value of *tattr* was not valid.

Setting Detach State

When a thread is created detached (PTHREAD_CREATE_DETACHED), its thread *ID* and other resources can be reused as soon as the thread exits. Use pthread_attr_setdetachstate(3C) when the calling thread does not want to wait for the thread to exit.

pthread_attr_setdetachstate(3C) Syntax

```
int pthread_attr_setdetachstate(pthread_attr_t *tattr,int  detachstate);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
/* set the thread detach state */
ret = pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
```

When a thread is created nondetached with PTHREAD_CREATE_JOINABLE, the assumption is that your application will wait for the thread to complete. That is, the program will execute a pthread_join() on the thread.

Whether a thread is created detached or nondetached, the process does not exit until all threads have exited. See "Finishing Up" on page 45 for a discussion of process termination caused by premature exit from main().

Note – When no explicit synchronization prevents a newly created, detached thread from exiting, its thread ID can be reassigned to another new thread before its creator returns from pthread_create().

Nondetached threads must have a thread join with the nondetached thread after the nondetached thread terminates. Otherwise, the resources of that thread are not released for use by new threads that commonly results in a memory leak. So, when you do not want a thread to be joined, create the thread as a detached thread.

EXAMPLE 3–1 Creating a Detached Thread

```
#include <pthread.h>

pthread_attr_t tattr;
pthread_t tid;
void *start_routine;
void arg
int ret;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);
ret = pthread_attr_setdetachstate (&tattr,PTHREAD_CREATE_DETACHED);
ret = pthread_create (&tid, &tattr, start_routine, arg);
```

pthread_attr_setdetachstate Return Values

pthread_attr_setdetachstate() returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL

Description: Indicates that the value of *detachstate* or *tattr* was not valid.

Getting the Detach State

Use pthread_attr_getdetachstate(3C) to retrieve the thread create state, which can be either detached or joined.

pthread_attr_getdetachstate Syntax

```
int pthread_attr_getdetachstate(const pthread_attr_t *tattr, int *detachstate;
```

```
#include <pthread.h>
pthread_attr_t tattr;
int detachstate;
int ret;
/* get detachstate of thread */
ret = pthread_attr_getdetachstate (&tattr, &detachstate);
```

pthread_attr_getdetachstate Return Values

`pthread_attr_getdetachstate()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Indicates that the value of *detachstate* is `NULL` or *tattr* is invalid.

Setting the Stack Guard Size

`pthread_attr_setguardsize`(3C) sets the *guardsize* of the *attr* object.

pthread_attr_setguardsize(3C) Syntax

```
#include <pthread.h>
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

The *guardsize* attribute is provided to the application for two reasons:

- Overflow protection can potentially result in wasted system resources. When your application creates a large number of threads, and you know that the threads will never overflow their stack, you can turn off guard areas. By turning off guard areas, you can conserve system resources.
- When threads allocate large data structures on stack, a large guard area might be needed to detect stack overflow.

The *guardsize* argument provides protection against overflow of the stack pointer. If a thread's stack is created with guard protection, the implementation allocates extra memory at the overflow end of the stack. This extra memory acts as a buffer against stack overflow of the stack pointer. If an application overflows into this buffer an error results, possibly in a `SIGSEGV` signal being delivered to the thread.

If *guardsize* is zero, a guard area is not provided for threads that are created with *attr*. If *guardsize* is greater than zero, a guard area of at least size *guardsize* bytes is provided for each thread created with *attr*. By default, a thread has an implementation-defined, nonzero guard area.

A conforming implementation is permitted to round up the value contained in *guardsize* to a multiple of the configurable system variable `PAGESIZE` . See `PAGESIZE` in `sys/mman.h`. If an implementation rounds up the value of *guardsize* to a multiple of `PAGESIZE`, a call to `pthread_attr_getguardsize()` that specifies *attr* stores, in *guardsize*, the guard size specified in the previous call to `pthread_attr_setguardsize()`.

pthread_attr_setguardsize Return Values

`pthread_attr_setguardsize()` fails if:

`EINVAL`

Description: The argument *attr* is invalid, the argument *guardsize* is invalid, or the argument *guardsize* contains an invalid value.

Getting the Stack Guard Size

`pthread_attr_getguardsize`(3C) gets the *guardsize* of the *attr* object.

pthread_attr_getguardsize Syntax

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
size_t  *restrict guardsize);
```

A conforming implementation is permitted to round up the value contained in *guardsize* to a multiple of the configurable system variable `PAGESIZE` . See `PAGESIZE` in `sys/mman.h`. If an implementation rounds up the value of *guardsize* to a multiple of `PAGESIZE`, a call to `pthread_attr_getguardsize()` that specifies *attr* stores, in *guardsize*, the guard size specified in the previous call to `pthread_attr_setguardsize()`.

pthread_attr_getguardsize Return Values

`pthread_attr_getguardsize()` fails if:

`EINVAL`

Description: The argument *attr* is invalid, the argument *guardsize* is invalid, or the argument *guardsize* contains an invalid value.

Setting the Scope

Use `pthread_attr_setscope`(3C) to establish the contention scope of a thread, either `PTHREAD_SCOPE_SYSTEM` or `PTHREAD_SCOPE_PROCESS`. With `PTHREAD_SCOPE_SYSTEM`, this thread contends with all threads in the system. With `PTHREAD_SCOPE_PROCESS` , this thread contends with other threads in the process.

Note – Both thread types are accessible only within a given process.

---

pthread_attr_setscope Syntax

```
int pthread_attr_setscope(pthread_attr_t *tattr,int scope);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int ret;

/* bound thread */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);

/* unbound thread */
ret = pthread_attr_setscope(&tattr, PTHREAD_SCOPE_PROCESS);
```

This example uses three function calls: a call to initialize the attributes, a call to set any variations from the default attributes, and a call to create the pthreads.

```
#include <pthread.h>

pthread_attr_t attr;
pthread_t tid;
void *start_routine(void *);
void *arg;
int ret;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);

ret =  pthread_attr_setscope(&tattr, PTHREAD_SCOPE_SYSTEM);
ret = pthread_create (&tid, &tattr, start_routine, arg);
```

pthread_attr_setscope Return Values

`pthread_attr_setscope()` returns zero after completing *successfully*. Any other return value indicates that an error occurred. If the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

Description: An attempt was made to set *tattr* to a value that is not valid.

Getting the Scope

Use `pthread_attr_getscope(3C)` to retrieve the thread scope.

pthread_attr_getscope Syntax

```
int pthread_attr_getscope(pthread_attr_t *restrict tattr, int *restrict scope);
```

```
#include <pthread.h>

pthread_attr_t tattr;
int scope;
int ret;

/* get scope of thread */
ret = pthread_attr_getscope(&tattr, &scope);
```

pthread_attr_getscope Return Values

`pthread_attr_getscope()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value of *scope* is `NULL` or *tattr* is invalid.

Setting the Thread Concurrency Level

`pthread_setconcurrency`(3C) is provided for standards compliance. `pthread_setconcurrency()` is used by an application to inform the system of the application's desired concurrency level. For the threads implementation introduced in the Solaris 9 release, this interface has no effect, all runnable threads are attached to LWPs.

pthread_setconcurrency Syntax

```
#include <pthread.h>

int pthread_setconcurrency(int new_level);
```

pthread_setconcurrency Return Values

`pthread_setconcurrency()` fails if the following conditions occur:

`EINVAL`

Description: The value specified by *new_level* is negative.

`EAGAIN`

Description: The value specified by *new_level* would cause a system resource to be exceeded.

Getting the Thread Concurrency Level

`pthread_getconcurrency`(3C) returns the value set by a previous call to `pthread_setconcurrency()`.

pthread_getconcurrency Syntax

```
#include <pthread.h>

int pthread_getconcurrency(void);
```

If the `pthread_setconcurrency()` function was not previously called, `pthread_getconcurrency()` returns zero.

pthread_getconcurrency Return Values

`pthread_getconcurrency()` always returns the concurrency level set by a previous call to `pthread_setconcurrency()`. If `pthread_setconcurrency()` has never been called, `pthread_getconcurrency()` returns zero.

Setting the Scheduling Policy

Use `pthread_attr_setschedpolicy`(3C) to set the scheduling policy. The POSIX standard specifies the scheduling policy values of `SCHED_FIFO` (first-in-first-out), `SCHED_RR` (round-robin), or `SCHED_OTHER` (an implementation-defined method). In the Oracle Solaris OS, `SCHED_OTHER` threads run in the traditional time-sharing (TS) scheduling class.

pthread_attr_setschedpolicy(3C) Syntax

```
int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int policy;
int ret;

/* set the scheduling policy to SCHED_OTHER */
ret = pthread_attr_setschedpolicy(&tattr, SCHED_OTHER);
```

- `SCHED_FIFO`

  A First-In-First-Out thread runs in the real-time (RT) scheduling class and require the calling process to be privileged. Such a thread, if not preempted by a higher priority thread, executes until it yields or blocks.

- `SCHED_RR`

  Round-Robin threads whose contention scope is system (`PTHREAD_SCOPE_SYSTEM`) are in real-time (`RT`) scheduling class if the calling process has an effective user id of `0`. These threads, if not preempted by a higher priority thread, and if the threads do not yield or block, will execute for the system-determined time period. Use `SCHED_RR` for threads that have a contention scope of process (`PTHREAD_SCOPE_PROCESS`) is based on the `TS` scheduling class. Additionally, the calling process for these threads does not have an effective userid of `0`.

A Round-Robin thread runs in the real-time (RT) scheduling class and requires the calling process to be privileged. If a round robin thread is not preempted by a higher priority thread, and does not yield or block, it will execute for a system-determined time period. The thread is then forced to yield to another real time thread of equal priority.

`SCHED_FIFO` and `SCHED_RR` are optional in the POSIX standard, and are supported for real-time threads only.

For a discussion of scheduling, see the section "Thread Scheduling" on page 23.

pthread_attr_setschedpolicy Return Values

`pthread_attr_setschedpolicy()` returns zero after completing successfully. Any other return value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: An attempt was made to set *tattr* to a value that is not valid.

`ENOTSUP`

Description: An attempt was made to set the attribute to an unsupported value.

Getting the Scheduling Policy

Use `pthread_attr_getschedpolicy(3C)` to retrieve the scheduling policy.

pthread_attr_getschedpolicy Syntax

```
int pthread_attr_getschedpolicy(pthread_attr_t *restrict tattr,
            int *restrict policy);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int policy;
int ret;

/* get scheduling policy of thread */
ret = pthread_attr_getschedpolicy (&tattr, &policy);
```

pthread_attr_getschedpolicy Return Values

`pthread_attr_getschedpolicy()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The parameter *policy* is `NULL` or *tattr* is invalid.

Setting the Inherited Scheduling Policy

Use pthread_attr_setinheritsched(3C) to set the inherited scheduling policy.

pthread_attr_setinheritsched Syntax

```
int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inheritsched);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int inheritsched;
int ret;

/* use  creating thread's scheduling policy and priority*/
ret = pthread_attr_setinheritsched(&tattr, PTHREAD_INHERIT_SCHED);
```

An *inheritsched* value of `PTHREAD_INHERIT_SCHED` means that the scheduling policy and priority of the creating thread are to be used for the created thread. The scheduling policy and priority in the attribute structure are to be ignored. An *inheritsched* value of `PTHREAD_EXPLICIT_SCHED` means that the scheduling policy and priority from the attribute structure are to be used for the created thread. The caller must have sufficient privilege for `pthread_create()` to succeed in this case.

pthread_attr_setinheritsched Return Values

`pthread_attr_setinheritsched()` returns zero after completing successfully. Any other return value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: An attempt was made to set *tattr* to a value that is not valid.

Getting the Inherited Scheduling Policy

pthread_attr_getinheritsched(3C) returns the *inheritsched* attribute contained in the attribute structure.

pthread_attr_getinheritsched Syntax

```
int pthread_attr_getinheritsched(pthread_attr_t *restrict tattr
       int *restrict inheritsched);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int inheritsched;
int ret;

ret = pthread_attr_getinheritsched (&tattr, &inheritsched);
```

pthread_attr_getinheritsched Return Values

`pthread_attr_getinheritsched()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The parameter *inherit* is `NULL` or *tattr* is invalid.

Setting the Scheduling Parameters

`pthread_attr_setschedparam`(3C) sets the scheduling parameters.

pthread_attr_setschedparam Syntax

```
int pthread_attr_setschedparam(pthread_attr_t *restrict tattr,
         const struct sched_param *restrict param);
```

```
#include <pthread.h>
pthread_attr_t tattr;
int ret;
int newprio;
sched_param param;
newprio = 30;
/* set the priority; others are unchanged */
param.sched_priority = newprio;
/* set the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);
```

Scheduling parameters are defined in the *param* structure. Only the priority parameter is supported.

pthread_attr_setschedparam Return Values

`pthread_attr_setschedparam()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following conditions occur, the function fails and returns the corresponding value.

`EINVAL`

Description: The value of *param* is `NULL` or *tattr* is invalid.

You can manage pthreads priority in either of two ways:

- You can set the priority attribute before creating a child thread
- You can change the priority of the parent thread and then change the priority back

Getting the Scheduling Parameters

`pthread_attr_getschedparam(3C)` returns the scheduling parameters defined by `pthread_attr_setschedparam()`.

pthread_attr_getschedparam Syntax

```
int pthread_attr_getschedparam(pthread_attr_t *restrict tattr,
       const struct sched_param *restrict param);
```

```
#include <pthread.h>
pthread_attr_t attr;
struct sched_param param;
int ret;
/* get the existing scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);
```

Creating a Thread With a Specified Priority

You can set the priority attribute before creating the thread. The child thread is created with the new priority that is specified in the `sched_param` structure. This structure also contains other scheduling information.

Example of Creating a Prioritized Thread

Example 3–2 shows an example of creating a child thread with a priority that is different from its parent's priority.

EXAMPLE 3–2 Creating a Prioritized Thread

```
#include <pthread.h>
#include <sched.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;
int newprio = 20;
sched_param param;

/* initialized with default attributes */
ret = pthread_attr_init (&tattr);

/* safe to get existing scheduling param */
ret = pthread_attr_getschedparam (&tattr, &param);

/* set the priority; others are unchanged */
param.sched_priority = newprio;

/* setting the new scheduling param */
ret = pthread_attr_setschedparam (&tattr, &param);

/* specify explicit scheduling */
```

EXAMPLE 3–2 Creating a Prioritized Thread *(Continued)*

```
ret = pthread_attr_setinheritsched (&tattr, PTHREAD_EXPLICIT_SCHED);

/* with new priority specified */
ret = pthread_create (&tid, &tattr, func, arg);
```

pthread_attr_getschedparam Return Values

`pthread_attr_getschedparam()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value of *param* is `NULL` or *tattr* is invalid.

About Stacks

Typically, thread stacks begin on page boundaries. Any specified size is rounded up to the next page boundary. A page with no access permission is appended to the overflow end of the stack. Most stack overflows result in sending a `SIGSEGV` signal to the offending thread. Thread stacks allocated by the caller are used without modification.

When a stack is specified, the thread should also be created with `PTHREAD_CREATE_JOINABLE`. That stack cannot be freed until the `pthread_join(3C)` call for that thread has returned. The thread's stack cannot be freed until the thread has terminated. The only reliable way to know if such a thread has terminated is through `pthread_join(3C)`.

Allocating Stack Space for Threads

Generally, you do not need to allocate stack space for threads. The system allocates 1 megabyte (for 32 bit systems) or 2 megabytes (for 64 bit systems) of virtual memory for each thread's stack with no swap space reserved. The system uses the `MAP_NORESERVE` option of `mmap()` to make the allocations.

Each thread stack created by the system has a red zone. The system creates the red zone by appending a page to the overflow end of a stack to catch stack overflows. This page is invalid and causes a memory fault if accessed. Red zones are appended to all automatically allocated stacks whether the size is specified by the application or the default size is used.

Note – Runtime stack requirements vary for library calls and dynamic linking. You should be absolutely certain that the specified stack satisfies the runtime requirements for library calls and dynamic linking.

Very few occasions exist when specifying a stack, its size, or both, is appropriate. Even an expert has a difficult time knowing whether the right size was specified. Even a program that is compliant with ABI standards cannot determine its stack size statically. The stack size is dependent on the needs of the particular runtime environment in execution.

Building Your Own Stack

When you specify the thread stack size, you must account for the allocations needed by the invoked function and by each subsequent function called. The accounting should include calling sequence needs, local variables, and information structures.

Occasionally, you want a stack that differs a bit from the default stack. An obvious situation is when the thread needs more than the default stack size. A less obvious situation is when the default stack is too large. You might be creating thousands of threads with insufficient virtual memory to handle the gigabytes of stack space required by thousands of default stacks.

The limits on the maximum size of a stack are often obvious, but what about the limits on its minimum size? Sufficient stack space must exist to handle all stack frames that are pushed onto the stack, along with their local variables, and so on.

To get the absolute minimum limit on stack size, call the macro PTHREAD_STACK_MIN. The PTHREAD_STACK_MIN macro returns the amount of required stack space for a thread that executes a NULL procedure. Useful threads need more than the minimum stack size, so be very careful when reducing the stack size.

Setting the Stack Size

pthread_attr_setstacksize(3C) sets the thread stack size.

pthread_attr_setstacksize Syntax

```
int pthread_attr_setstacksize(pthread_attr_t *tattr, size_t size);
```

```
#include <pthread.h>
#include <limits.h>
pthread_attr_t tattr;
size_t size;
int ret;
size = (PTHREAD_STACK_MIN + 0x4000);
/* setting a new size */
ret = pthread_attr_setstacksize(&tattr, size);
```

The *size* attribute defines the size of the stack (in bytes) that the system allocates. The *size* should not be less than the system-defined minimum stack size. See "About Stacks" on page 65 for more information.

size contains the number of bytes for the stack that the new thread uses. If *size* is zero, a default size is used. In most cases, a zero value works best.

PTHREAD_STACK_MIN is the amount of stack space that is required to start a thread. This stack space does not take into consideration the threads routine requirements that are needed to execute application code.

EXAMPLE 3–3 Example of Setting Stack Size

```
#include <pthread.h>
#include <limits.h>

pthread_attr_t tattr;
pthread_t tid;
int ret;

size_t size = PTHREAD_STACK_MIN + 0x4000;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the size of the stack also */
ret = pthread_attr_setstacksize(&tattr, size);

/* only size specified in tattr*/
ret = pthread_create(&tid, &tattr, start_routine, arg);
```

pthread_attr_setstacksize Return Values

pthread_attr_setstacksize() returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL

Description: The value of *size* is less than PTHREAD_STACK_MIN, or exceeds a system-imposed limit, or *tattr* is not valid.

Getting the Stack Size

pthread_attr_getstacksize(3C) returns the stack size set by pthread_attr_setstacksize().

pthread_attr_getstacksize Syntax

```
int pthread_attr_getstacksize(pthread_attr_t *restrict tattr, size_t *restrict size);
```

```
#include <pthread.h>
pthread_attr_t tattr;
size_t size;
int ret;
/* getting the stack size */
ret = pthread_attr_getstacksize(&tattr, &size);
```

pthread_attr_getstacksize Return Values

`pthread_attr_getstacksize()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: *tattr* or *size* is not valid.

Setting the Stack Address and Size

`pthread_attr_setstack`(3C) sets the thread stack address and size.

pthread_attr_setstack(3C) Syntax

```
int pthread_attr_setstack(pthread_attr_t *tattr,void *stackaddr, size_t stacksize);
```

```
#include <pthread.h>
#include <limits.h>
pthread_attr_t tattr;
void *base;
size_t size;
int ret;
base = (void *) malloc(PTHREAD_STACK_MIN + 0x4000);
/* setting a new address and size */
ret = pthread_attr_setstack(&tattr, base,PTHREAD_STACK_MIN + 0x4000);
```

The *stackaddr* attribute defines the base (low address) of the thread's stack. The *stacksize* attribute specifies the size of the stack. If *stackaddr* is set to non-null, rather than the `NULL` default, the system initializes the stack at that address, assuming the size to be *stacksize*.

base contains the address for the stack that the new thread uses. If *base* is `NULL`, then `pthread_create(3C)` allocates a stack for the new thread with at least *stacksize* bytes.

pthread_attr_setstack(3C) Return Values

`pthread_attr_setstack()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value of *base* or *tattr* is incorrect. The value of *stacksize* is less than `PTHREAD_STACK_MIN`.

The following example shows how to create a thread with a custom stack address and size.

```
#include <pthread.h>

pthread_attr_t tattr;
```

```
pthread_t tid;
int ret;
void *stackbase;
size_t size;

/* initialized with default attributes */
ret = pthread_attr_init(&tattr);

/* setting the base address and size of the stack */
ret = pthread_attr_setstack(&tattr, stackbase,size);

/* address and size specified */
ret = pthread_create(&tid, &tattr, func, arg);
```

Getting the Stack Address and Size

`pthread_attr_getstack(3C)` returns the thread stack address and size set by `pthread_attr_setstack()`.

pthread_attr_getstack Syntax

```
int pthread_attr_getstack(pthread_attr_t *restrict tattr,
         void **restrict stackaddr, size_t *restrict stacksize);
```

```
#include <pthread.h>

pthread_attr_t tattr;
void *base;
size_t size;
int ret;

/* getting a stack address and size */
ret = pthread_attr_getstack (&tattr
, &base, &size);
```

pthread_attr_getstack Return Values

`pthread_attr_getstack()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value of *tattr* is incorrect.

CHAPTER 4

Programming with Synchronization Objects

This chapter describes the synchronization types that are available with threads. The chapter also discusses when and how to use synchronization.

- "Mutual Exclusion Lock Attributes" on page 72
- "Using Mutual Exclusion Locks" on page 85
- "Using Spin Locks" on page 98
- "Condition Variable Attributes" on page 102
- "Using Condition Variables" on page 107
- "Synchronization With Semaphores" on page 119
- "Read-Write Lock Attributes" on page 126
- "Using Read-Write Locks" on page 129
- "Using Barrier Synchronization" on page 137
- "Synchronization Across Process Boundaries" on page 142
- "Comparing Primitives" on page 143

Synchronization objects are variables in memory that you access just like data. Threads in different processes can communicate with each other through synchronization objects that are placed in threads-controlled shared memory. The threads can communicate with each other even though the threads in different processes are generally invisible to each other.

Synchronization objects can also be placed in files. The synchronization objects can have lifetimes beyond the life of the creating process.

The available types of synchronization objects are

- Mutex locks
- Condition variables
- Read-Write locks
- Semaphores

Situations that can benefit from the use of synchronization include the following:

- Synchronization is the only way to ensure consistency of shared data.

- Threads in two or more processes can use a single synchronization object jointly. Because reinitializing a synchronization object sets the object to the *unlocked* state, the synchronization object should be initialized by only one of the cooperating processes.
- Synchronization can ensure the safety of mutable data.
- A process can map a file and direct a thread in this process get a record's lock. Once the lock is acquired, any thread in any process mapping the file attempting to acquire the lock is blocked until the lock is released.
- Accessing a single primitive variable, such as an integer, can use more than one memory cycle for a single memory load. More than one memory cycle is used where the integer is not aligned to the bus data width or is larger than the data width. While integer misalignment cannot happen on the SPARC architecture, portable programs cannot rely on the proper alignment.

Note – On 32-bit architectures, a `long long` is not atomic. (An *atomic* operation cannot be divided into smaller operations.) A `long long` is read and written as two 32-bit quantities. The types `int`, `char` , `float`, and pointers are atomic on SPARC architecture machines and Intel Architecture machines.

Mutual Exclusion Lock Attributes

Use mutual exclusion locks (mutexes) to serialize thread execution. Mutual exclusion locks synchronize threads, usually by ensuring that only one thread at a time executes a critical section of code. Mutex locks can also preserve single-threaded code.

To change the default mutex attributes, you can declare and initialize an attribute object. Often, the mutex attributes are set in one place at the beginning of the application so the attributes can be located quickly and modified easily. Table 4–1 lists the functions that manipulate mutex attributes.

TABLE 4–1 Mutex Attributes Routines

Operation	Related Function Description
Initialize a mutex attribute object	"`pthread_mutexattr_init` Syntax" on page 73
Destroy a mutex attribute object	"`pthread_mutexattr_destroy` Syntax" on page 75
Set the scope of a mutex	"`pthread_mutexattr_setpshared` Syntax" on page 75
Get the scope of a mutex	"`pthread_mutexattr_getpshared` Syntax" on page 76
Set the mutex type attribute	"`pthread_mutexattr_settype` Syntax" on page 76
Get the mutex type attribute	"`pthread_mutexattr_gettype` Syntax" on page 78

TABLE 4–1 Mutex Attributes Routines *(Continued)*

Operation	Related Function Description
Set mutex attribute's protocol	"pthread_mutexattr_setprotocol Syntax" on page 78
Get mutex attribute's protocol	"pthread_mutexattr_getprotocol Syntax" on page 80
Set mutex attribute's priority ceiling	"pthread_mutexattr_setprioceiling Syntax" on page 80
Get mutex attribute's priority ceiling	"pthread_mutexattr_getprioceiling Syntax" on page 81
Set mutex's priority ceiling	"pthread_mutex_setprioceiling Syntax" on page 82
Get mutex's priority ceiling	"pthread_mutex_getprioceiling Syntax" on page 82
Set mutex's robust attribute	"pthread_mutexattr_setrobust_np Syntax" on page 83
Get mutex's robust attribute	"pthread_mutexattr_getrobust_np Syntax" on page 85

Initializing a Mutex Attribute Object

Use pthread_mutexattr_init(3C) to initialize attributes that are associated with the mutex object to their default values. Storage for each attribute object is allocated by the threads system during execution.

pthread_mutexattr_init Syntax

```
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);
```

```
#include <pthread.h>

pthread_mutexattr_t mattr;
int ret;

/* initialize an attribute to default value */
ret = pthread_mutexattr_init(&mattr);
```

mattr is an opaque type that contains a system-allocated attribute object. See Table 4–2 for information about the attributes in the *mattr* object.

Before a mutex attribute object can be reinitialized, the object must first be destroyed by a call to pthread_mutexattr_destroy(3C). The pthread_mutexattr_init() call results in the allocation of an opaque object. If the object is not destroyed, a memory leak results.

TABLE 4–2 Default Attribute Values for *mattr*

Attribute	Value	Result
pshared	`PTHREAD_PROCESS_PRIVATE`	The initialized mutex can be used within a process. Only those threads created by the same process can operate on the mutex.
type	`PTHREAD_MUTEX_DEFAULT`	The Oracle Solaris Pthreads implementation maps `PTHREAD_MUTEX_DEFAULT` to `PTHREAD_MUTEX_NORMAL`, which does not detect deadlock.
protocol	`PTHREAD_PRIO_NONE`	Thread priority and scheduling are not affected by the priority of the mutex owned by the thread.
prioceiling	–	The *prioceiling* value is drawn from the existing priority range for the `SCHED_FIFO` policy, as returned by the `sched_get_priority_min()` and `sched_get_priority_max()` functions. This priority range is determined by the Oracle Solaris version on which the mutex is created.
robustness	`PTHREAD_MUTEX_STALLED_NP`	When the owner of a mutex dies, all future calls to `pthread_mutex_lock()` for this mutex will be blocked from progress.

pthread_mutexattr_init Return Values

`pthread_mutexattr_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. If either of the following conditions occurs, the function fails and returns the corresponding value.

`ENOMEM`

Description: Insufficient memory exists to initialize the mutex attribute object.

Destroying a Mutex Attribute Object

pthread_mutexattr_destroy(3C) deallocates the storage space used to maintain the attribute object created by pthread_mutexattr_init().

pthread_mutexattr_destroy Syntax

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr)

#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
/* destroy an attribute */
ret = pthread_mutexattr_destroy(&mattr);
```

pthread_mutexattr_destroy Return Values

pthread_mutexattr_destroy() returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

EINVAL

Description: The value specified by *mattr* is invalid.

Setting the Scope of a Mutex

pthread_mutexattr_setpshared(3C) sets the scope of the mutex variable.

pthread_mutexattr_setpshared Syntax

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *restrict mattr,
         int *restrict pshared);

#include <pthread.h>
pthread_mutexattr_t mattr;
int ret;
ret = pthread_mutexattr_init(&mattr);
/* * resetting to its default value: private */
ret = pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_PRIVATE);
```

The scope of a mutex variable can be either process private (intraprocess) or system wide (interprocess). To share the mutex among threads from more than one process, create the mutex in shared memory with the *pshared* attribute set to PTHREAD_PROCESS_SHARED .

If the mutex *pshared* attribute is set to PTHREAD_PROCESS_PRIVATE , only those threads created by the same process can operate on the mutex.

pthread_mutexattr_setpshared Return Values

`pthread_mutexattr_setpshared()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value specified by *mattr* is invalid.

Getting the Scope of a Mutex

`pthread_mutexattr_getpshared`(3C) returns the scope of the mutex variable defined by `pthread_mutexattr_setpshared()`.

pthread_mutexattr_getpshared Syntax

```
int pthread_mutexattr_getpshared(pthread_mutexattr_t *restrict mattr,
          int *restrict pshared);
```

```
#include <pthread.h>
pthread_mutexattr_t mattr;
int pshared, ret;
/* get pshared of mutex */
ret = pthread_mutexattr_getpshared(&mattr, &pshared);
```

Get the current value of *pshared* for the attribute object *mattr*. The value is either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

pthread_mutexattr_getpshared Return Values

`pthread_mutexattr_getpshared()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value specified by *mattr* is invalid.

Setting the Mutex Type Attribute

`pthread_mutexattr_settype`(3C) sets the mutex *type* attribute.

pthread_mutexattr_settype Syntax

```
#include <pthread.h>

int pthread_mutexattr_settype(pthread_mutexattr_t  *attr , int type);
```

The default value of the *type* attribute is `PTHREAD_MUTEX_DEFAULT`.

The *type* argument specifies the type of mutex. The following list describes the valid mutex types:

`PTHREAD_MUTEX_NORMAL`

Description: This type of mutex does not detect deadlock. A thread attempting to relock this mutex without first unlocking the mutex deadlocks. Attempting to unlock a mutex locked by a different thread results in undefined behavior. Attempting to unlock an unlocked mutex results in undefined behavior.

`PTHREAD_MUTEX_ERRORCHECK`

Description: This type of mutex provides error checking. A thread attempting to relock this mutex without first unlocking the mutex returns an error. A thread attempting to unlock a mutex that another thread has locked returns an error. A thread attempting to unlock an unlocked mutex returns an error.

`PTHREAD_MUTEX_RECURSIVE`

Description: A thread attempting to relock this mutex without first unlocking the mutex succeeds in locking the mutex. The relocking deadlock that can occur with mutexes of type `PTHREAD_MUTEX_NORMAL` cannot occur with this type of mutex. Multiple locks of this mutex require the same number of unlocks to release the mutex before another thread can acquire the mutex. A thread attempting to unlock a mutex that another thread has locked returns an error. A thread attempting to unlock an unlocked mutex returns an error.

`PTHREAD_MUTEX_DEFAULT`

Description: An implementation is allowed to map this attribute to one of the other mutex types. The Oracle Solaris implementation maps this attribute to `PTHREAD_PROCESS_NORMAL`.

pthread_mutexattr_settype Return Values

If successful, the `pthread_mutexattr_settype` function returns zero. Otherwise, an error number is returned to indicate the error.

`EINVAL`

Description: The value *type* or *attr* is invalid.

Getting the Mutex Type Attribute

`pthread_mutexattr_gettype`(3C) gets the mutex *type* attribute set by `pthread_mutexattr_settype()`.

pthread_mutexattr_gettype Syntax

```
#include <pthread.h>

int pthread_mutexattr_gettype(pthread_mutexattr_t  *restrict attr ,
int  *restrict type);
```

The default value of the *type* attribute is `PTHREAD_MUTEX_DEFAULT`.

The *type* argument specifies the type of mutex. Valid mutex types include

- `PTHREAD_MUTEX_NORMAL`
- `PTHREAD_MUTEX_ERRORCHECK`
- `PTHREAD_MUTEX_RECURSIVE`
- `PTHREAD_MUTEX_DEFAULT`

For a description of each type, see “`pthread_mutexattr_settype` Syntax” on page 76.

pthread_mutexattr_gettype Return Values

On successful completion, `pthread_mutexattr_gettype()` returns 0. Any other return value indicates that an error occurred.

`EINVAL`
  Description: The value specified by *type* is invalid.

Setting the Mutex Attribute's Protocol

`pthread_mutexattr_setprotocol`(3C) sets the protocol attribute of a mutex attribute object.

pthread_mutexattr_setprotocol Syntax

```
#include <pthread.h>
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
          int protocol);
```

attr points to a mutex attribute object created by an earlier call to `pthread_mutexattr_init()`.

protocol defines the protocol that is applied to the mutex attribute object.

The value of *protocol* that is defined in `pthread.h` can be one of the following values: `PTHREAD_PRIO_NONE` , `PTHREAD_PRIO_INHERIT`, or `PTHREAD_PRIO_PROTECT` .

- `PTHREAD_PRIO_NONE`

  A thread's priority and scheduling are not affected by the mutex ownership.
- `PTHREAD_PRIO_INHERIT`

This protocol value affects an owning thread's priority and scheduling. When higher-priority threads block on one or more mutexes owned by `thrd1` where those mutexes are initialized with `PTHREAD_PRIO_INHERIT`, `thrd1` runs with the higher of its priority or the highest priority of any thread waiting on any of the mutexes owned by `thrd1`.

If `thrd1` blocks on a mutex owned by another thread, `thrd3`, the same priority inheritance effect recursively propagates to `thrd3`.

Use `PTHREAD_PRIO_INHERIT` to avoid priority inversion. Priority inversion occurs when a low-priority thread holds a lock that a higher-priority thread requires. The higher-priority thread cannot continue until the lower-priority thread releases the lock.

Without priority inheritance, the lower priority thread might not be scheduled to run for a long time, causing the higher priority thread to block equally long. Priority inheritance temporarily raises the priority of the lower priority thread so it will be scheduled to run quickly and release the lock, allowing the higher priority thread to acquire it. The lower-priority thread reverts to its lower priority when it releases the lock.

- `PTHREAD_PRIO_PROTECT`

  This protocol value affects the priority and scheduling of a thread, such as `thrd2`, when the thread owns one or more mutexes that are initialized with `PTHREAD_PRIO_PROTECT`. `thrd2` runs with the higher of its priority or the highest-priority ceiling of all mutexes owned by `thrd2`. Higher-priority threads blocked on any of the mutexes, owned by `thrd2`, have no effect on the scheduling of `thrd2`.

The `PTHREAD_PRIO_INHERIT` and `PTHREAD_PRIO_PROTECT` mutex attributes are usable only by privileged processes running in the realtime (RT) scheduling class `SCHED_FIFO` or `SCHED_RR`.

A thread can simultaneously own several mutexes initialized with a mix of `PTHREAD_PRIO_INHERIT` and `PTHREAD_PRIO_PROTECT`. In this case, the thread executes at the highest priority obtained by either of these protocols.

pthread_mutexattr_setprotocol Return Values

On successful completion, `pthread_mutexattr_setprotocol()` returns 0. Any other return value indicates that an error occurred.

If either of the following conditions occurs, `pthread_mutexattr_setprotocol()` might fail and return the corresponding value.

`EINVAL`

Description: The value specified by *attr* or *protocol* is not valid.

`EPERM`

Description: The caller does not have the privilege to perform the operation.

Getting the Mutex Attribute's Protocol

pthread_mutexattr_getprotocol(3C) gets the protocol attribute of a mutex attribute object.

pthread_mutexattr_getprotocol Syntax

```
#include <pthread.h>
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict attr,
int *restrict protocol);
```

attr points to a mutex attribute object created by an earlier call to `pthread_mutexattr_init()`.

protocol contains one of the following protocol attributes: `PTHREAD_PRIO_NONE`, `PTHREAD_PRIO_INHERIT`, or `PTHREAD_PRIO_PROTECT` which are defined by the header `<pthread.h>`.

pthread_mutexattr_getprotocol Return Values

On successful completion, `pthread_mutexattr_getprotocol()` returns 0. Any other return value indicates that an error occurred.

If either of the following conditions occurs, `pthread_mutexattr_getprotocol()` might fail and return the corresponding value.

`EINVAL`

Description: The value specified by *attr* is NULL, or the value specified by *attr* or *protocol* is invalid.

`EPERM`

Description: The caller does not have the privilege to perform the operation.

Setting the Mutex Attribute's Priority Ceiling

pthread_mutexattr_setprioceiling(3C) sets the priority ceiling attribute of a mutex attribute object.

pthread_mutexattr_setprioceiling Syntax

```
#include <pthread.h>
int pthread_mutexattr_setprioceiling(pthread_mutexatt_t *attr, int prioceiling);
```

attr points to a mutex attribute object created by an earlier call to `pthread_mutexattr_init()`.

prioceiling specifies the priority ceiling of initialized mutexes. The ceiling defines the minimum priority level at which the critical section guarded by the mutex is executed. *prioceiling* falls within the maximum range of priorities defined by `SCHED_FIFO`. To avoid priority inversion, set *prioceiling* to a priority higher than or equal to the highest priority of all threads that might lock the particular mutex.

pthread_mutexattr_setprioceiling Return Values

On successful completion, `pthread_mutexattr_setprioceiling()` returns 0. Any other return value indicates that an error occurred.

If either of the following conditions occurs, `pthread_mutexattr_setprioceiling()` might fail and return the corresponding value.

`EINVAL`

Description: The value specified by *attr* is NULL or invalid or *prioceiling* is invalid.

`EPERM`

Description: The caller does not have the privilege to perform the operation.

Getting the Mutex Attribute's Priority Ceiling

`pthread_mutexattr_getprioceiling`(3C) gets the priority ceiling attribute of a mutex attribute object.

pthread_mutexattr_getprioceiling Syntax

```
#include <pthread.h>
int pthread_mutexattr_getprioceiling(const pthread_mutexatt_t *restrict attr,
           int *restrict prioceiling);
```

attr designates the attribute object created by an earlier call to `pthread_mutexattr_init()`.

`pthread_mutexattr_getprioceiling()` returns the priority ceiling of initialized mutexes in *prioceiling*. The ceiling defines the minimum priority level at which the critical section guarded by the mutex is executed. *prioceiling* falls within the maximum range of priorities defined by `SCHED_FIFO`. To avoid priority inversion, set *prioceiling* to a priority higher than or equal to the highest priority of all threads that might lock the particular mutex.

pthread_mutexattr_getprioceiling Return Values

On successful completion, `pthread_mutexattr_getprioceiling()` returns 0. Any other return value indicates that an error occurred.

If either of the following conditions occurs, `pthread_mutexattr_getprioceiling()` might fail and return the corresponding value.

`EINVAL`

Description: The value specified by *attr* is NULL.

`EPERM`

Description: The caller does not have the privilege to perform the operation.

Setting the Mutex's Priority Ceiling

pthread_mutexattr_setprioceiling(3C) sets the priority ceiling of a mutex.

pthread_mutex_setprioceiling Syntax

```
#include <pthread.h>
int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex,
          int prioceiling, int *restrict old_ceiling);
```

pthread_mutex_setprioceiling() changes the priority ceiling, *prioceiling*, of a mutex, *mutex*. pthread_mutex_setprioceiling() locks a mutex if unlocked, or blocks until pthread_mutex_setprioceiling() successfully locks the mutex, changes the priority ceiling of the mutex and releases the mutex. The process of locking the mutex need not adhere to the priority protect protocol.

If pthread_mutex_setprioceiling() succeeds, the previous value of the priority ceiling is returned in *old_ceiling*. If pthread_mutex_setprioceiling() fails, the mutex priority ceiling remains unchanged.

pthread_mutex_setprioceiling Return Values

On successful completion, pthread_mutex_setprioceiling() returns 0. Any other return value indicates that an error occurred.

If any of the following conditions occurs, pthread_mutex_setprioceiling() might fail and return the corresponding value.

EINVAL

Description: The priority requested by *prioceiling* is out of range.

EINVAL

Description: The mutex was not initialized with its *protocol* attribute having the value of THREAD_PRIO_PROTECT.

EPERM

Description: The caller does not have the privilege to perform the operation.

Getting the Mutex's Priority Ceiling

pthread_mutexattr_getprioceiling(3C) gets the priority ceiling of a mutex.

pthread_mutex_getprioceiling Syntax

```
#include <pthread.h>
int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex,
          int *restrict prioceiling);
```

`pthread_mutex_getprioceiling()` returns the priority ceiling, *prioceiling* of a *mutex*.

pthread_mutex_getprioceiling Return Values

On successful completion, `pthread_mutex_getprioceiling()` returns 0. Any other return value indicates that an error occurred.

If any of the following conditions occurs, `pthread_mutexatt_getprioceiling()` fails and returns the corresponding value.

If any of the following conditions occurs, `pthread_mutex_getprioceiling()` might fail and return the corresponding value.

`EINVAL`

Description: The value specified by *mutex* does not refer to a currently existing mutex.

`EPERM`

Description: The caller does not have the privilege to perform the operation.

Setting the Mutex's Robust Attribute

`pthread_mutexattr_setrobust_np` sets the robust attribute of a mutex attribute object.

pthread_mutexattr_setrobust_np Syntax

```
#include <pthread.h>
int pthread_mutexattr_setrobust_np(pthread_mutexattr_t *attr, int *robustness);
```

Note – `pthread_mutexattr_setrobust_np()` applies only if the symbol `_POSIX_THREAD_PRIO_INHERIT` is defined.

In the Oracle Solaris 10 and prior releases, the `PTHREAD_MUTEX_ROBUST_NP` attribute can only be applied to mutexes that are also marked with the `PTHREAD_PRIO_INHERIT` protocol attribute. This restriction is lifted in subsequent Oracle Solaris releases.

---

attr points to the mutex attribute object previously created by a call to `pthread_mutexattr_init()`.

robustness defines the behavior when the owner of the mutex terminates without unlocking the mutex, usually because its process terminated abnormally. The value of *robustness* that is defined in `pthread.h` is `PTHREAD_MUTEX_ROBUST_NP` or `PTHREAD_MUTEX_STALLED_NP`. The default value is `PTHREAD_MUTEX_STALLED_NP` .

- `PTHREAD_MUTEX_STALLED_NP`

When the owner of the mutex terminates without unlocking the mutex, all subsequent calls to `pthread_mutex_lock()` are blocked from progress in an unspecified manner.

- `PTHREAD_MUTEX_ROBUST_NP`

  When the owner of the mutex terminates without unlocking the mutex, the mutex is unlocked. The next owner of this mutex acquires the mutex with an error return of `EOWNERDEAD`.

Note – Your application must always check the return code from `pthread_mutex_lock()` for a mutex initialized with the `PTHREAD_MUTEX_ROBUST_NP` attribute.

- The new owner of this mutex should make the state protected by the mutex consistent. This state might have been left inconsistent when the previous owner terminated.
- If the new owner is able to make the state consistent, call `pthread_mutex_consistent_np()` for the mutex before unlocking the mutex. This marks the mutex as consistent and subsequent calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()` will behave in the normal manner.
- If the new owner is *not* able to make the state consistent, do *not* call `pthread_mutex_consistent_np()` for the mutex, but unlock the mutex.

  All waiters are awakened and all subsequent calls to `pthread_mutex_lock()` fail to acquire the mutex. The return code is `ENOTRECOVERABLE`. The mutex can be made consistent by calling `pthread_mutex_destroy()` to uninitialize the mutex, and calling `pthread_mutex_int()` to reinitialize the mutex. However, the state that was protected by the mutex remains inconsistent and some form of application recovery is required.

If the thread that acquires the lock with `EOWNERDEAD` terminates without unlocking the mutex, the next owner acquires the lock with an `EOWNERDEAD` return code.

pthread_mutexattr_setrobust_np Return Values

On successful completion, `pthread_mutexattr_setrobust_np()` returns 0. Any other return value indicates that an error occurred.

`pthread_mutexattr_setrobust_np()` might fail if the following condition occurs:

`EINVAL`

Description: The value specified by *attr* or *robustness* is invalid.

Getting the Mutex's Robust Attribute

`pthread_mutexattr_getrobust_np` gets the robust attribute of a mutex attribute object.

pthread_mutexattr_getrobust_np Syntax

```
#include <pthread.h>
int pthread_mutexattr_getrobust_np(const pthread_mutexattr_t *attr, int *robustness);
```

attr points to the mutex attribute object previously created by a call to `pthread_mutexattr_init()`.

robustness is the value of the robust attribute of a mutex attribute object.

pthread_mutexattr_getrobust_np Return Values

On successful completion, `pthread_mutexattr_getrobust_np()` returns 0. Any other return value indicates that an error occurred.

`pthread_mutexattr_getrobust_np()` might fail if the following condition occurs:

`EINVAL`

Description: The value specified by *attr* or *robustness* is invalid.

Using Mutual Exclusion Locks

Table 4–3 lists the functions that manipulate mutex locks.

TABLE 4–3 Routines for Mutual Exclusion Locks

Operation	Related Function Description
Initialize a mutex	"`pthread_mutex_init` Syntax" on page 86
Make mutex consistent	"`pthread_mutex_consistent_np` Syntax" on page 87
Lock a mutex	"`pthread_mutex_lock` Syntax" on page 88
Unlock a mutex	"`pthread_mutex_unlock` Syntax" on page 89
Lock with a nonblocking mutex	"`pthread_mutex_trylock` Syntax" on page 90
Lock a mutex before a specified time	"`pthread_mutex_timedlock()` Syntax" on page 91
Lock a mutex within a specified time interval	"`pthread_mutex_reltimedlock_np()` Syntax" on page 92
Destroy a mutex	"`pthread_mutex_destroy` Syntax" on page 93

The default scheduling policy, `SCHED_OTHER`, does not specify the order in which threads can acquire a lock. When multiple `SCHED_OTHER` threads are waiting for a mutex, the order of acquisition is undefined. Under the `SCHED_FIFO` and `SCHED_RR` real-time scheduling policies, the behavior is to unblock waiting threads in priority order.

Initializing a Mutex

Use `pthread_mutex_init`(3C) to initialize the mutex pointed at by *mp* to its default value or to specify mutex attributes that have already been set with `pthread_mutexattr_init()`. The default value for *mattr* is `NULL`.

pthread_mutex_init Syntax

```
int pthread_mutex_init(pthread_mutex_t *restrict mp,
          const pthread_mutexattr_t *restrict mattr);

#include <pthread.h>

pthread_mutex_t mp = PTHREAD_MUTEX_INITIALIZER;
pthread_mutexattr_t mattr;
int ret;

/* initialize a mutex to its default value */
ret = pthread_mutex_init(&mp, NULL);

/* initialize a mutex */
ret = pthread_mutex_init(&mp, &mattr);
```

When the mutex is initialized, the mutex is in an unlocked state. The mutex can be in memory that is shared between processes or in memory private to a process.

Note – For a mutex that is being initialized with the `PTHREAD_MUTEX_ROBUST_NP` attribute, the mutex memory must be cleared to zero before initialization.

The effect of *mattr* set to `NULL` is the same as passing the address of a default mutex attribute object, but without the memory overhead.

Use the macro `PTHREAD_MUTEX_INITIALIZER` to initialize statically defined mutexes to their default attributes.

Do not reinitialize or destroy a mutex lock while other threads are using the mutex. Program failure results if either action is not done correctly. If a mutex is reinitialized or destroyed, the application must be sure the mutex is not currently in use.

pthread_mutex_init Return Values

`pthread_mutex_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

EBUSY

Description: The implementation has detected an attempt to reinitialize the object referenced by *mp*, a previously initialized but not yet destroyed mutex.

EINVAL

Description: The *mattr* attribute value is invalid. The mutex has not been modified.

EFAULT

Description: The address for the mutex pointed at by *mp* is invalid.

Making a Mutex Consistent

If the owner of a robust mutex terminates without unlocking the mutex, the mutex is unlocked and marked inconsistent. The next owner acquires the lock with an EOWNERDEAD return code.

pthread_mutex_consistent_np() makes the mutex object, *mutex*, consistent after the death of its owner.

pthread_mutex_consistent_np Syntax

```
#include <pthread.h>
int pthread_mutex_consistent_np(pthread_mutex_t *mutex);
```

Call pthread_mutex_lock() to acquire the inconsistent mutex. The EOWNWERDEAD return value indicates an inconsistent mutex.

Call pthread_mutex_consistent_np() while holding the mutex acquired by a previous call to pthread_mutex_lock().

The critical section protected by the mutex might have been left in an inconsistent state by a failed owner. In this case, make the mutex consistent only if you can make the critical section protected by the mutex consistent.

Calls to pthread_mutex_lock(), pthread_mutex_unlock() , and pthread_mutex_trylock() for a consistent mutex behave in the normal manner.

The behavior of pthread_mutex_consistent_np() for a mutex that is *not* inconsistent, or is not held, is undefined.

pthread_mutex_consistent_np Return Values

pthread_mutex_consistent_np() returns zero after completing successfully. Any other return value indicates that an error occurred.

pthread_mutex_consistent_np() fails if the following condition occurs:

EINVAL

Description: The current thread does not own the mutex or the mutex is not a `PTHREAD_MUTEX_ROBUST_NP` mutex having an inconsistent state.

Locking a Mutex

Use `pthread_mutex_lock`(3C) to lock the mutex pointed to by *mutex*.

pthread_mutex_lock Syntax

```
int  pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
#include <pthread.h>

pthread_mutex_t mutex;
int ret;

ret = pthread_ mutex_lock(&mp); /* acquire the mutex */
```

When `pthread_mutex_lock()` returns, the mutex is locked. The calling thread is the owner. If the mutex is already locked and owned by another thread, the calling thread blocks until the mutex becomes available.

If the mutex type is `PTHREAD_MUTEX_NORMAL` , deadlock detection is not provided. Attempting to relock the mutex causes deadlock. If a thread attempts to unlock a mutex not locked by the thread or a mutex that is unlocked, undefined behavior results.

If the mutex type is `PTHREAD_MUTEX_ERRORCHECK` , then error checking is provided. If a thread attempts to relock a mutex that the thread has already locked, an error is returned. If a thread attempts to unlock a mutex not locked by the thread or a mutex that is unlocked, an error is returned.

If the mutex type is `PTHREAD_MUTEX_RECURSIVE` , then the mutex maintains the concept of a lock count. When a thread successfully acquires a mutex for the first time, the lock count is set to 1. Every time a thread relocks this mutex, the lock count is incremented by 1. Every time the thread unlocks the mutex, the lock count is decremented by 1. When the lock count reaches 0, the mutex becomes available for other threads to acquire. If a thread attempts to unlock a mutex not locked by the thread or a mutex that is unlocked, an error is returned.

The mutex type `PTHREAD_MUTEX_DEFAULT` is the same as `PTHREAD_MUTEX_NORMAL`.

pthread_mutex_lock Return Values

`pthread_mutex_lock()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

EAGAIN
: **Description:** The mutex could not be acquired because the maximum number of recursive locks for mutex has been exceeded.

EDEADLK
: **Description:** The current thread already owns the mutex.

If the mutex was initialized with the `PTHREAD_MUTEX_ROBUST_NP`robustness attribute, `pthread_mutex_lock()` may return one of the following values:

EOWNERDEAD
: **Description:** The last owner of this mutex terminated while holding the mutex. This mutex is now owned by the caller. The caller must attempt to make the state protected by the mutex consistent.

  If the caller is able to make the state consistent, call `pthread_mutex_consistent_np()` for the mutex and unlock the mutex. Subsequent calls to `pthread_mutex_lock()` behave normally.

  If the caller is unable to make the state consistent, do not call `pthread_mutex_init()` for the mutex. Unlock the mutex instead. Subsequent calls to `pthread_mutex_lock()` fail to acquire the mutex and return an `ENOTRECOVERABLE` error code.

  If the owner that acquired the lock with `EOWNERDEAD` terminates while holding the mutex, the next owner acquires the lock with `EOWNERDEAD`.

ENOTRECOVERABLE
: **Description:** The mutex you are trying to acquire was protecting state left irrecoverable by the mutex's previous owner. The mutex has not been acquired. This irrecoverable condition can occur when:

  - The lock was previously acquired with `EOWNERDEAD`
  - The owner was unable to cleanup the state
  - The owner unlocked the mutex without making the mutex state consistent

ENOMEM
: **Description:** The limit on the number of simultaneously held mutexes has been exceeded.

Unlocking a Mutex

Use `pthread_mutex_unlock(3C)` to unlock the mutex pointed to by *mutex*.

pthread_mutex_unlock Syntax

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
#include <pthread.h>

pthread_mutex_t mutex;
int ret;

ret = pthread_mutex_unlock(&mutex); /* release the mutex */
```

`pthread_mutex_unlock()` releases the mutex object referenced by *mutex*. The manner in which a mutex is released is dependent upon the mutex's type attribute. If threads are blocked on the *mutex* object when `pthread_mutex_unlock()` is called and the mutex becomes available, the scheduling policy determines which thread acquires the mutex. For `PTHREAD_MUTEX_RECURSIVE` mutexes, the mutex becomes available when the count reaches zero and the calling thread no longer has any locks on this mutex.

pthread_mutex_unlock Return Values

`pthread_mutex_unlock()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EPERM`
  Description: The current thread does not own the mutex.

Locking a Mutex Without Blocking

Use `pthread_mutex_trylock`(3C) to attempt to lock the mutex pointed to by *mutex*, and return immediately if the mutex is already locked.

pthread_mutex_trylock Syntax

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
#include <pthread.h>

pthread_mutex_t mutex;
int ret;

ret = pthread_mutex_trylock(&mutex); /* try to lock the mutex */
```

`pthread_mutex_trylock()` is a nonblocking version of `pthread_mutex_lock()`. If the mutex object referenced by *mutex* is currently locked by any thread, including the current thread, the call returns immediately. Otherwise, the mutex is locked and the calling thread is the owner.

pthread_mutex_trylock Return Values

`pthread_mutex_trylock()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

EBUSY

Description: The mutex could not be acquired because the mutex pointed to by *mutex* was already locked.

EAGAIN

Description: The mutex could not be acquired because the maximum number of recursive locks for *mutex* has been exceeded.

If the symbol _POSIX_THREAD_PRIO_INHERIT is defined, the mutex is initialized with the protocol attribute value PTHREAD_PRIO_INHERIT. Additionally, if the *robustness* argument of pthread_mutexattr_setrobust_np() is PTHREAD_MUTEX_ROBUST_NP, the function fails and returns one of the following values:

EOWNERDEAD

Description: See the discussion in "pthread_mutex_lock Return Values" on page 88.

ENOTRECOVERABLE

Description: See the discussion in "pthread_mutex_lock Return Values" on page 88.

ENOMEM

Description: The limit on the number of simultaneously held mutexes has been exceeded.

Locking a Mutex Before a Specified Absolute Time

Use the pthread_mutex_timedlock(3C) function to attempt until a specified time to lock a mutex object.

This function works as the pthread_mutex_lock() function does, except that it does not block indefinitely. If the mutex is already locked, the calling thread is blocked until the mutex becomes available, but only until the timeout is reached. If the timeout occurs before the mutex becomes available, the function returns.

pthread_mutex_timedlock() Syntax

```
int  pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
          const struct timespec *restrict abs_timeout);

#include <pthread.h>
#include <time.h>

pthread_mutex_t mutex;
timestruct_t abs_timeout;
int ret;

ret = pthread_mutex_timedlock(&mutex,  &abs_timeout);
```

`pthread_mutex_timedlock()` Return Values

The `pthread_mutex_timedlock()` function return 0 if it locks the mutex successfully. Otherwise, an error number is returned to indicate the error.

`EINVAL`

Description: The mutex was created with the protocol attribute having the value `PTHREAD_PRIO_PROTECT` and the calling thread's priority is higher than the mutex's current priority ceiling.

Description: The value specified by *mutex* does not refer to an initialized mutex object.

Description: The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than 0 or greater than or equal to 1000 million.

`ETIMEDOUT`

Description: The mutex could not be locked before the specified timeout expired.

See the discussion in "`pthread_mutex_lock` Return Values" on page 88.

Locking a Mutex Within a Specified Time Interval

Use the `pthread_mutex_reltimedlock_np(3C)` function to attempt until a specified amount of time elapses to lock a mutex object.

The timeout expires when the time interval specified by *rel_timeout* passes, as measured by the `CLOCK_REALTIME` clock, or if the time interval specified by *rel_timeout* is negative at the time of the call.

`pthread_mutex_reltimedlock_np()` Syntax

```
int  pthread_mutex_reltimedlock_np(pthread_mutex_t *restrict mutex,
          const struct timespec *restrict rel_timeout);

#include <pthread.h>
#include <time.h>

pthread_mutex_t mutex;
timestruct_t rel_timeout;
int ret;

ret = pthread_mutex_reltimedlock_np(&mutex,  &rel_timeout);
```

`pthread_mutex_reltimedlock_np()` Return Values

The `pthread_mutex_reltimedlock_np()` function returns 0 if it locks the mutex successfully. Otherwise, an error number is returned to indicate the error.

EINVAL

Description: The mutex was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than the mutex's current priority ceiling.

Description: The value specified by *mutex* does not refer to an initialized mutex object.

Description: The process or thread would have blocked, and the *abs_timeout* parameter specified a nanoseconds field value less than 0 or greater than or equal to 1000 million.

ETIMEDOUT

Description: The mutex could not be locked before the specified timeout expired.

See the discussion in "pthread_mutex_lock Return Values" on page 88.

Destroying a Mutex

Use pthread_mutex_destroy(3C) to destroy any state that is associated with the mutex pointed to by *mp* .

pthread_mutex_destroy Syntax

```
int pthread_mutex_destroy(pthread_mutex_t *mp);
```

```
#include <pthread.h>

pthread_mutex_t mp;
int ret;

ret = pthread_mutex_destroy(&mp); /* mutex is destroyed */
```

Note that the space for storing the mutex is not freed.

pthread_mutex_destroy Return Values

pthread_mutex_destroy() returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occur, the function fails and returns the corresponding value.

EINVAL

Description: The value specified by *mp* does not refer to an initialized mutex object.

Code Examples of Mutex Locking

Example 4–1 shows some code fragments with mutex locking.

EXAMPLE 4–1 Mutex Lock Example

```
#include <pthread.h>

pthread_mutex_t count_mutex;
long long count;

void
increment_count()
{
    pthread_mutex_lock(&count_mutex);
    count = count + 1;
    pthread_mutex_unlock(&count_mutex);
}

long long
get_count()
{
    long long c;

    pthread_mutex_lock(&count_mutex);
    c = count;
    pthread_mutex_unlock(&count_mutex);
    return (c);
}
```

The two functions in Example 4–1 use the mutex lock for different purposes. The `increment_count()` function uses the mutex lock to ensure an atomic update of the shared variable. The `get_count()` function uses the mutex lock to guarantee that the 64-bit quantity *count* is read atomically. On a 32-bit architecture, a `long long` is really two 32-bit quantities.

When you read an integer value, the operation is atomic because an integer is the common word size on most machines.

Examples of Using Lock Hierarchies

Occasionally, you might want to access two resources at once. Perhaps you are using one of the resources, and then discover that the other resource is needed as well. A problem exists if two threads attempt to claim both resources but lock the associated mutexes in different orders. For example, if the two threads lock mutexes 1 and 2 respectively, a deadlock occurs when each attempts to lock the other mutex. Example 4–2 shows possible deadlock scenarios.

EXAMPLE 4–2 Deadlock

Thread 1	Thread 2
`pthread_mutex_lock(&m1);`	`pthread_mutex_lock(&m2);`
`/* use resource 1 */`	`/* use resource 2 */`
`pthread_mutex_lock(&m2);`	`pthread_mutex_lock(&m1);`
`/* use resources 1 and 2 */`	`/* use resources 1 and 2 */`
`pthread_mutex_unlock(&m2);`	`pthread_mutex_unlock(&m1);`
`pthread_mutex_unlock(&m1);`	`pthread_mutex_unlock(&m2);`

The best way to avoid this problem is to make sure that when threads lock multiple mutexes, the threads do so in the same order. When locks are always taken in a prescribed order, deadlock should not occur. This technique, known as lock hierarchies, orders the mutexes by logically assigning numbers to the mutexes.

Also, honor the restriction that you cannot take a mutex that is assigned *n* when you are holding any mutex assigned a number that is greater than *n*.

However, this technique cannot always be used. Sometimes, you must take the mutexes in an order other than prescribed. To prevent deadlock in such a situation, use `pthread_mutex_trylock()`. One thread must release its mutexes when the thread discovers that deadlock would otherwise be inevitable.

EXAMPLE 4–3 Conditional Locking

thread1	thread2
`pthread_mutex_lock(&m1);` `pthread_mutex_lock(&m2);`	`for (; ;)`
/* no processing */	`{ pthread_mutex_lock(&m2);`
`pthread_mutex_unlock(&m2);`	`if(pthread_mutex_trylock(&m1)==0)`
`pthread_mutex_unlock(&m1);`	/* got it */
	`break;`
	/* didn't get it */
	`pthread_mutex_unlock(&m2);`
	`}`
	/* get locks; no processing */
	`pthread_mutex_unlock(&m1);`
	`pthread_mutex_unlock(&m2);`

In Example 4–3, thread1 locks mutexes in the prescribed order, but thread2 takes the mutexes out of order. To make certain that no deadlock occurs, thread2 has to take mutex1 very carefully. If thread2 blocks while waiting for the mutex to be released, thread2 is likely to have just entered into a deadlock with thread1.

To ensure that thread2 does not enter into a deadlock, thread2 calls `pthread_mutex_trylock()`, which takes the mutex if available. If the mutex is not available, thread2 returns immediately, reporting failure. At this point, thread2 must release mutex2. Thread1 can now lock mutex2, and then release both mutex1 and mutex2.

Examples of Using Nested Locking With a Singly-Linked List

Example 4–4 and Example 4–5 show how to take three locks at once. Deadlock is prevented by taking the locks in a prescribed order.

EXAMPLE 4–4 Singly-Linked List Structure

```
typedef struct node1 {
    int value;
    struct node1 *link;
    pthread_mutex_t lock;
} node1_t;

node1_t ListHead;
```

This example uses a singly linked list structure with each node that contains a mutex. To remove a node from the list, first search the list starting at ListHead until the desired node is found. ListHead is never removed.

To protect this search from the effects of concurrent deletions, lock each node before any of its contents are accessed. Because all searches start at ListHead, a deadlock cannot occur because the locks are always taken in list order.

When the desired node is found, lock both the node and its predecessor since the change involves both nodes. Because the predecessor's lock is always taken first, you are again protected from deadlock. Example 4–5 shows the C code to remove an item from a singly-linked list.

EXAMPLE 4–5 Singly-Linked List With Nested Locking

```
node1_t *delete(int value)
{
    node1_t *prev, *current;

    prev = &ListHead;
    pthread_mutex_lock(&prev->lock);
    while ((current = prev->link) != NULL) {
        pthread_mutex_lock(&current->lock);
        if (current->value == value) {
            prev->link = current->link;
            pthread_mutex_unlock(&current->lock);
            pthread_mutex_unlock(&prev->lock);
            current->link = NULL;
            return(current);
        }
        pthread_mutex_unlock(&prev->lock);
        prev = current;
    }
    pthread_mutex_unlock(&prev->lock);
    return(NULL);
}
```

Example of Nested Locking With a Circularly-Linked List

Example 4–6 modifies the previous list structure by converting the list structure into a circular list. Because a distinguished head node no longer exists, a thread can be associated with a particular node and can perform operations on that node and its neighbor Lock hierarchies do not work easily here because the obvious hierarchy, following the links, is circular.

EXAMPLE 4–6 Circular-Linked List Structure

```
typedef struct node2 {
    int value;
    struct node2 *link;
    pthread_mutex_t lock;
} node2_t;
```

Here is the C code that acquires the locks on two nodes and performs an operation that involves both locks.

EXAMPLE 4–7 Circular Linked List With Nested Locking

```
void Hit Neighbor(node2_t *me) {
    while (1) {
        pthread_mutex_lock(&me->lock);
        if (pthread_mutex_trylock(&me->link->lock)!= 0) {
            /* failed to get lock */
            pthread_mutex_unlock(&me->lock);
            continue;
        }
        break;
    }
    me->link->value += me->value;
    me->value /=2;
    pthread_mutex_unlock(&me->link->lock);
    pthread_mutex_unlock(&me->lock);
}
```

Using Spin Locks

Spin locks are a low-level synchronization mechanism suitable primarily for use on shared memory multiprocessors. When the calling thread requests a spin lock that is already held by another thread, the second thread spins in a loop to test if the lock has become available. When the lock is obtained, it should be held only for a short time, as the spinning wastes processor cycles. Callers should unlock spin locks before calling sleep operations to enable other threads to obtain the lock.

Spin locks can be implemented using mutexes and conditional variables, but the `pthread_spin_*` functions are a standardized way to practice spin locking. The `pthread_spin_*` functions require much lower overhead for locks of short duration.

When performing any lock, a trade-off is made between the processor resources consumed while setting up to block the thread and the processor resources consumed by the thread while it is blocked. Spin locks require few resources to set up the blocking of a thread and then do a simple loop, repeating the atomic locking operation until the lock is available. The thread continues to consume processor resources while it is waiting.

Compared to spin locks, mutexes consume a larger amount of processor resources to block the thread. When a mutex lock is not available, the thread changes its scheduling state and adds itself to the queue of waiting threads. When the lock becomes available, these steps must be reversed before the thread obtains the lock. While the thread is blocked, it consumes no processor resources.

Therefore, spin locks and mutexes can be useful for different purposes. Spin locks might have lower overall overhead for very short-term blocking, and mutexes might have lower overall overhead when a thread will be blocked for longer periods of time.

Initializing a Spin Lock

Use the pthread_spin_init(3C) function to allocate resources required to use a spin lock, and initialize the lock to an unlocked state.

pthread_spin_init() Syntax

```
int  pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```

```
#include <pthread.h>

pthread_spinlock_t lock;
int pshared;
int ret;

/* initialize a spin lock */
ret = pthread_spin_init(&lock, pshared);
```

The *pshared* attribute has one of the following values:

PTHREAD_PROCESS_SHARED

Description: Permits a spin lock to be operated on by any thread that has access to the memory where the spin lock is allocated. Operation on the lock is permitted even if the lock is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE

Description: Permits a spin lock to be operated upon only by threads created within the same process as the thread that initialized the spin lock. If threads of differing processes attempt to operate on such a spin lock, the behavior is undefined. The default value of the process-shared attribute is PTHREAD_PROCESS_PRIVATE.

pthread_spin_init() Return Values

Upon successful completion, the pthread_spin_init() function returns 0. Otherwise, one of the following error codes is returned.

EAGAIN

Description: The system lacks the necessary resources to initialize another spin lock.

EBUSY

Description: The system has detected an attempt to initialize or destroy a spin lock while the lock is in use (for example, while being used in a pthread_spin_lock() call) by another thread.

EINVAL

Description: The value specified by *lock* is invalid.

Acquiring a Spin Lock

Use the `pthread_spin_lock`(3C) to lock a spin lock. The calling thread acquires the lock if it is not held by another thread. Otherwise, the thread does not return from the `pthread_spin_lock()` call until the lock becomes available. The results are undefined if the calling thread holds the lock at the time the call is made.

`pthread_spin_lock()` Syntax

```
int  pthread_spin_lock(pthread_spinlock_t *lock);
```

```
#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_ spin_lock(&lock); /* lock the spinlock */
```

`pthread_spin_lock()` Return Values

Upon successful completion, the `pthread_spin_lock()` function returns 0. Otherwise, one of the following error codes is returned.

`EDEADLK`

Description: The current thread already owns the spin lock.

`EINVAL`

Description: The value specified by *lock* does not refer to an initialized spin lock object.

Acquiring a Non-Blocking Spin Lock

Use the `pthread_spin_trylock`(3C) function to lock a spin lock and fail immediately if the lock is held by another thread.

`pthread_spin_trylock()` Syntax

```
int  pthread_spin_trylock(pthread_spinlock_t *lock);
```

```
#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_spin_trylock(&lock); /* try to lock the spin lock */
```

`pthread_spin_trylock()` Return Values

Upon successful completion, the `pthread_spin_trylock()` function returns 0. Otherwise, one of the following error codes is returned.

`EBUSY`

Description: A thread currently owns the spin lock.

`EINVAL`

Description: The value specified by *lock* does not refer to an initialized spin lock object.

Unlocking a Spin Lock

Use the `pthread_spin_unlock`(3C) function to release a locked spin lock.

`pthread_spin_unlock()` Syntax

```
int  pthread_spin_unlock(pthread_spinlock_t *lock);
```

```
#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_spin_unlock(&lock); /* spinlock is unlocked */
```

`pthread_spin_unlock()` Return Values

Upon successful completion, the `pthread_spin_unlock()` function returns 0. Otherwise, one of the following error codes is returned.

`EPERM`

Description: The calling thread does not hold the lock.

`EINVAL`

Description: The value specified by *lock* does not refer to an initialized spin lock object.

Destroying a Spin Lock

Use the `pthread_spin_destroy`(3C) function to destroy a spin lock and release any resources used by the lock.

`pthread_spin_destroy()` Syntax

```
int  pthread_spin_destroy(pthread_spinlock_t *lock);
```

```
#include <pthread.h>

pthread_spinlock_t lock;
int ret;

ret = pthread_spin_destroy(&lock); /* spinlock is destroyed */
```

The effect of subsequent use of the lock is undefined until the lock is reinitialized by another call to `pthread_spin_init()`. The results are undefined if `pthread_spin_destroy()` is called when a thread holds the lock, or if this function is called with an uninitialized thread spin lock.

`pthread_spin_destroy()` Return Values

`EBUSY`

Description: The system has detected an attempt to initialize or destroy a spin lock while the lock is in use (for example, while being used in a `pthread_spin_lock()` call) by another thread.

`EINVAL`

Description: The value specified by *lock* is invalid.

Condition Variable Attributes

Use condition variables to atomically block threads until a particular condition is true. Always use condition variables together with a mutex lock.

With a condition variable, a thread can atomically block until a condition is satisfied. The condition is tested under the protection of a mutual exclusion lock (mutex).

When the condition is false, a thread usually blocks on a condition variable and atomically releases the mutex waiting for the condition to change. When another thread changes the condition, that thread can signal the associated condition variable to cause one or more waiting threads to perform the following actions:

- Wake up
- Acquire the mutex again
- Re-evaluate the condition

Condition variables can be used to synchronize threads among processes in the following situations:

- The threads are allocated in memory that can be written to
- The memory is shared by the cooperating processes

The scheduling policy determines how blocking threads are awakened. The default scheduling policy, `SCHED_OTHER`, does not specify the order in which threads are awakened. Under the `SCHED_FIFO` and `SCHED_RR` real-time scheduling policies, threads are awakened in priority order.

The attributes for condition variables must be set and initialized before the condition variables can be used. The functions that manipulate condition variable attributes are listed in Table 4–4.

TABLE 4–4 Condition Variable Attributes

Operation	Function Description
Initialize a condition variable attribute	"`pthread_condattr_init` Syntax" on page 103
Remove a condition variable attribute	"`pthread_condattr_destroy` Syntax" on page 104
Set the scope of a condition variable	"`pthread_condattr_setpshared` Syntax" on page 104
Get the scope of a condition variable	"`pthread_condattr_getpshared` Syntax" on page 105
Get the clock selection condition variable attribute	"`pthread_condattr_getclock` Syntax" on page 107
Set the clock selection condition variable attribute	"`pthread_condattr_setclock` Syntax" on page 106

Initializing a Condition Variable Attribute

Use `pthread_condattr_init`(3C) to initialize attributes that are associated with this object to their default values. Storage for each attribute object is allocated by the threads system during execution.

pthread_condattr_init Syntax

```
int pthread_condattr_init(pthread_condattr_t *cattr);
```

```
#include <pthread.h>
pthread_condattr_t cattr;
int ret;

/* initialize an attribute to default value */
ret = pthread_condattr_init(&cattr);
```

The default value of the *pshared* attribute when this function is called is `PTHREAD_PROCESS_PRIVATE`. This value of *pshared* means that the initialized condition variable can be used within a process.

cattr is an opaque data type that contains a system-allocated attribute object. The possible values of *cattr*'s scope are `PTHREAD_PROCESS_PRIVATE` and `PTHREAD_PROCESS_SHARED` . `PTHREAD_PROCESS_PRIVATE` is the default value.

Before a condition variable attribute can be reused, the attribute must first be reinitialized by `pthread_condattr_destroy(3C)`. The `pthread_condattr_init()` call returns a pointer to an opaque object. If the object is not destroyed, a memory leak results.

pthread_condattr_init Return Values

`pthread_condattr_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`ENOMEM`

Description: Insufficient memory allocated to initialize the thread attributes object.

`EINVAL`

Description: The value specified by *cattr* is invalid.

Removing a Condition Variable Attribute

Use `pthread_condattr_destroy`(3C) to remove storage and render the attribute object invalid.

pthread_condattr_destroy Syntax

```
int pthread_condattr_destroy(pthread_condattr_t *cattr);
```

```
#include <pthread.h>
pthread_condattr_t cattr;
int ret;

/* destroy an attribute */
ret
 = pthread_condattr_destroy(&cattr);
```

pthread_condattr_destroy Return Values

`pthread_condattr_destroy()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value specified by *cattr* is invalid.

Setting the Scope of a Condition Variable

`pthread_condattr_setpshared`(3C) sets the scope of a condition variable to either process private (intraprocess) or system wide (interprocess).

pthread_condattr_setpshared Syntax

```
int pthread_condattr_setpshared(pthread_condattr_t *cattr, int pshared);
```

```
#include <pthread.h>
pthread_condattr_t cattr;
int ret;

/* all processes */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

/* within a process */
ret = pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_PRIVATE);
```

A condition variable created with the *pshared* attribute set in shared memory to `PTHREAD_PROCESS_SHARED`, can be shared among threads from more than one process.

If the mutex *pshared* attribute is set to `PTHREAD_PROCESS_PRIVATE`, only those threads created by the same process can operate on the mutex. `PTHREAD_PROCESS_PRIVATE` is the default value. `PTHREAD_PROCESS_PRIVATE` behaves like a local condition variable. The behavior of `PTHREAD_PROCESS_SHARED` is equivalent to a global condition variable.

pthread_condattr_setpshared Return Values

`pthread_condattr_setpshared()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value of *cattr* is invalid, or the *pshared* value is invalid.

Getting the Scope of a Condition Variable

`pthread_condattr_getpshared`(3C) gets the current value of *pshared* for the attribute object *cattr*.

pthread_condattr_getpshared Syntax

```
int pthread_condattr_getpshared(const pthread_condattr_t *restrict cattr,
          int *restrict pshared);
```

```
#include <pthread.h>

pthread_condattr_t cattr;
int pshared;
int ret;

/* get pshared value of condition variable */
ret = pthread_condattr_getpshared(&cattr, &pshared);
```

The value of the attribute object is either `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`.

pthread_condattr_getpshared Return Values

`pthread_condattr_getpshared()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value of *cattr* is invalid.

Setting the Clock Selection Condition Variable

Use the `pthread_condattr_setclock`(3C) function to set the clock attribute in an initialized attributes object referenced by *attr*. If `pthread_condattr_setclock()` is called with a *clock_id* argument that refers to a CPU-time clock, the call fails. The clock attribute is the clock ID of the clock that is used to measure the timeout service of `pthread_cond_timedwait()`. The default value of the clock attribute refers to the system clock, `CLOCK_REALTIME`. At this time, the only other possible value for the clock attribute is `CLOCK_MONOTONIC`.

pthread_condattr_setclock Syntax

```
int pthread_condattr_setclock(pthread_condattr_t attr,
          clockid_t clock_id);

#include <pthread.h>

pthread_condattr_t attr
clockid_t clock_id
int ret


ret = pthread_condattr_setclock(&attr &clock_id
```

pthread_condattr_setclock Returns

`pthread_condattr_setclock()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value specified by *clock_id* does not refer to a known clock, or is a CPU-time clock.

Getting the Clock Selection Condition Variable

Use the `pthread_condattr_getclock`(3C) function to obtain the value of the clock attribute from the attributes object referenced by *attr*. The clock attribute is the clock ID of the clock that is used to measure the timeout service of `pthread_cond_timedwait()`.

pthread_condattr_getclock Syntax

```
int pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
          clockid_t *restrict clock_id);

#include <pthread.h>

pthread_condattr_t attr
clockid_t clock_id
int ret


ret = pthread_condattr_getclock(&attr &clock_id
```

pthread_condattr_getclock Returns

`pthread_condattr_getclock()` returns zero after completing successfully and stores the value of the `clock` attribute of *attr* into the object referenced by the *clock_id* argument. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value of *attr* is invalid.

Using Condition Variables

This section explains how to use condition variables. Table 4–5 lists the functions that are available.

TABLE 4–5 Condition Variables Functions

Operation	Related Function Description
Initialize a condition variable	"`pthread_cond_init` Syntax" on page 108
Block on a condition variable	"`pthread_cond_wait` Syntax" on page 109
Unblock a specific thread	"`pthread_cond_signal` Syntax" on page 110
Block until a specified time	"`pthread_cond_timedwait` Syntax" on page 111
Block for a specified interval	"`pthread_cond_reltimedwait_np` Syntax" on page 113
Unblock all threads	"`pthread_cond_broadcast` Syntax" on page 114
Destroy condition variable state	"`pthread_cond_destroy` Syntax" on page 115

Initializing a Condition Variable

Use `pthread_cond_init(3C)` to initialize the condition variable pointed at by *cv* to its default value, or to specify condition variable attributes that are already set with `pthread_condattr_init()`.

pthread_cond_init Syntax

```
int pthread_cond_init(pthread_cond_t *restrict cv,
          const pthread_condattr_t *restrict cattr);

#include <pthread.h>

pthread_cond_t cv;
pthread_condattr_t cattr;
int ret;

/* initialize a condition variable to its default value */
ret = pthread_cond_init(&cv, NULL);

/* initialize a condition variable */
ret = pthread_cond_init(&cv, &cattr);
```

The effect of *cattr* set to `NULL` is the same as passing the address of a default condition variable attribute object, but without the memory overhead.

Use the macro `PTHREAD_COND_INITIALIZER` to initialize statically defined condition variables to their default attributes. The `PTHREAD_COND_INITIALIZER` macro has the same effect as dynamically allocating `pthread_cond_init()` with null attributes. No error checking is done.

Multiple threads must not simultaneously initialize or reinitialize the same condition variable. If a condition variable is reinitialized or is destroyed, the application must be sure that the condition variable is not in use.

pthread_cond_init Return Values

`pthread_cond_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value specified by *cattr* is invalid.

`EBUSY`

Description: The condition variable is being used.

`EAGAIN`

Description: The necessary resources are not available.

ENOMEM

Description: Insufficient memory exists to initialize the condition variable.

Blocking on a Condition Variable

Use pthread_cond_wait(3C) to atomically release the mutex pointed to by *mp* and to cause the calling thread to block on the condition variable pointed to by *cv*.

pthread_cond_wait Syntax

```
int pthread_cond_wait(pthread_cond_t *restrict cv,pthread_mutex_t *restrict mutex);
```

```
#include <pthread.h>

pthread_cond_t cv;
pthread_mutex_t mp;
int ret;

/* wait on condition variable */
ret = pthread_cond_wait(&cv, &
mp);
```

The blocked thread can be awakened by a pthread_cond_signal() , a pthread_cond_broadcast(), or when interrupted by delivery of a signal.

Any change in the value of a condition that is associated with the condition variable cannot be inferred by the return of pthread_cond_wait(). Such conditions must be reevaluated.

The pthread_cond_wait() routine always returns with the mutex locked and owned by the calling thread, even when returning an error.

This function blocks until the condition is signaled. The function atomically releases the associated mutex lock before blocking, and atomically acquires the mutex again before returning.

In typical use, a condition expression is evaluated under the protection of a mutex lock. When the condition expression is false, the thread blocks on the condition variable. The condition variable is then signaled by another thread when the thread changes the condition value. The change causes at least one thread that is waiting on the condition variable to unblock and to reacquire the mutex.

The condition that caused the wait must be retested before continuing execution from the point of the pthread_cond_wait(). The condition could change before an awakened thread reacquires the mutes and returns from pthread_cond_wait(). A waiting thread could be awakened spuriously. The recommended test method is to write the condition check as a while() loop that calls pthread_cond_wait().

```
    pthread_mutex_lock();
        while(condition_is_false)
            pthread_cond_wait();
    pthread_mutex_unlock();
```

The scheduling policy determines the order in which blocked threads are awakened. The default scheduling policy, SCHED_OTHER, does not specify the order in which threads are awakened. Under the SCHED_FIFO and SCHED_RR real-time scheduling policies, threads are awakened in priority order.

Note – pthread_cond_wait() is a cancellation point. If a cancel is pending and the calling thread has cancellation enabled, the thread terminates and begins executing its cleanup handlers while continuing to hold the lock.

---

pthread_cond_wait Return Values

pthread_cond_wait() returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

EINVAL

Description: The value specified by *cv* or *mp* is invalid.

Unblocking One Thread

Use pthread_cond_signal(3C) to unblock one thread that is blocked on the condition variable pointed to by *cv*.

pthread_cond_signal Syntax

```
int pthread_cond_signal(pthread_cond_t *cv);
```

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* one condition variable is signaled */
ret = pthread_cond_signal(&cv);
```

Modify the associated condition under the protection of the same mutex used with the condition variable being signaled. Otherwise, the condition could be modified between its test and blocking in pthread_cond_wait(), which can cause an infinite wait.

The scheduling policy determines the order in which blocked threads are awakened. The default scheduling policy, SCHED_OTHER, does not specify the order in which threads are awakened. Under the SCHED_FIFO and SCHED_RR real-time scheduling policies, threads are awakened in priority order.

When no threads are blocked on the condition variable, calling `pthread_cond_signal()` has no effect.

EXAMPLE 4–8 Using `pthread_cond_wait()` and `pthread_cond_signal()`

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;

decrement_count()
{
    pthread_mutex_lock(&count_lock);
    while (count == 0)
        pthread_cond_wait(&count_nonzero, &count_lock);
    count = count - 1;
    pthread_mutex_unlock(&count_lock);
}

increment_count()
{
    pthread_mutex_lock(&count_lock);
    if (count == 0)
        pthread_cond_signal(&count_nonzero);
    count = count + 1;
    pthread_mutex_unlock(&count_lock);
}
```

pthread_cond_signal Return Values

`pthread_cond_signal()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: *cv* points to an illegal address.

Example 4–8 shows how to use `pthread_cond_wait()` and `pthread_cond_signal()`.

Blocking Until a Specified Time

Use `pthread_cond_timedwait(3C)` as you would use `pthread_cond_wait()`, except that `pthread_cond_timedwait()` does not block past the time of day specified by *abstime* .

pthread_cond_timedwait Syntax

```
int pthread_cond_timedwait(pthread_cond_t *restrict cv,
          pthread_mutex_t *restrict mp,
          const struct timespec *restrict abstime);
```

```
#include <pthread.h>
#include <time.h>
```

```
pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t abstime;
int ret;

/* wait on condition variable */
ret = pthread_cond_timedwait(&cv, &
mp, &abstime);
```

`pthread_cond_timedwait()` always returns with the mutex locked and owned by the calling thread, even when `pthread_cond_timedwait()` is returning an error.

The `pthread_cond_timedwait()` function blocks until the condition is signaled or until the time of day specified by the last argument has passed.

Note – `pthread_cond_timedwait()` is also a cancellation point.

EXAMPLE 4–9 Timed Condition Wait

```
pthread_timestruc_t to;
pthread_mutex_t m;
pthread_cond_t c;
...
pthread_mutex_lock(&m);
clock_gettime(CLOCK_REALTIME, &to);
to.tv_sec += TIMEOUT;
while (cond == FALSE) {
    err = pthread_cond_timedwait(&c, &m, &to);
    if (err == ETIMEDOUT) {
        /* timeout, do something */
        break;
    }
}
pthread_mutex_unlock(&m);
```

pthread_cond_timedwait Return Values

`pthread_cond_timedwait()` returns zero after completing successfully. Any other return value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: *cv*, *mp*, or *abstime* points to an illegal address.

`EINVAL`

Description: Different mutexes were supplied for concurrent `pthread_cond_timedwait()` operations on the same condition variable.

`ETIMEDOUT`

Description: The time specified by *abstime* has passed.

EPERM

Description: The mutex was not owned by the current thread at the time of the call.

The timeout is specified as a time of day so that the condition can be retested efficiently without recomputing the value, as shown in Example 4–9.

Blocking For a Specified Interval

Use pthread_cond_reltimedwait_np(3C) as you would use pthread_cond_timedwait() with one exception. pthread_cond_reltimedwait_np() takes a relative time interval rather than an absolute future time of day as the value of its last argument.

pthread_cond_reltimedwait_np Syntax

```
int pthread_cond_reltimedwait_np(pthread_cond_t *cv,
           pthread_mutex_t *mp,
          const struct timespec *reltime);
```

```
#include <pthread.h>
#include <time.h>
pthread_cond_t cv;
pthread_mutex_t mp;
timestruct_t reltime;
int ret;

/* wait on condition variable */
ret = pthread_cond_reltimedwait_np(&cv, &mp, &reltime);
```

pthread_cond_reltimedwait_np() always returns with the mutex locked and owned by the calling thread, even when pthread_cond_reltimedwait_np() is returning an error. The pthread_cond_reltimedwait_np() function blocks until the condition is signaled or until the time interval specified by the last argument has elapsed.

Note – pthread_cond_reltimedwait_np() is also a cancellation point.

pthread_cond_reltimedwait_np Return Values

pthread_cond_reltimedwait_np() returns zero after completing successfully. Any other return value indicates that an error occurred. When either of the following conditions occurs, the function fails and returns the corresponding value.

EINVAL

Description: The value specified by *reltime* is invalid.

ETIMEDOUT

Description: The time interval specified by *reltime* has passed.

Unblocking All Threads

Use `pthread_cond_broadcast`(3C) to unblock all threads that are blocked on the condition variable pointed to by *cv*, specified by `pthread_cond_wait()`.

pthread_cond_broadcast Syntax

```
int pthread_cond_broadcast(pthread_cond_t *cv);
```

```
#include <pthread.h>

pthread_cond_t cv;
int ret;

/* all condition variables are signaled */
ret = pthread_cond_broadcast(&cv);
```

When no threads are blocked on the condition variable, `pthread_cond_broadcast()` has no effect.

Since `pthread_cond_broadcast()` causes all threads blocked on the condition to contend again for the mutex lock, use `pthread_cond_broadcast()` with care. For example, use `pthread_cond_broadcast()` to allow threads to contend for varying resource amounts when resources are freed, as shown in Example 4–10.

EXAMPLE 4–10 Condition Variable Broadcast

```
pthread_mutex_t rsrc_lock;
pthread_cond_t rsrc_add;
unsigned int resources;

get_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    while (resources < amount) {
        pthread_cond_wait(&rsrc_add, &rsrc_lock);
    }
    resources -= amount;
    pthread_mutex_unlock(&rsrc_lock);
}

add_resources(int amount)
{
    pthread_mutex_lock(&rsrc_lock);
    resources += amount;
    pthread_cond_broadcast(&rsrc_add);
    pthread_mutex_unlock(&rsrc_lock);
}
```

Note that in `add_resources()` whether *resources* are updated first, or if `pthread_cond_broadcast()` is called first inside the mutex lock does not matter.

Modify the associated condition under the protection of the same mutex that is used with the condition variable being signaled. Otherwise, the condition could be modified between its test and blocking in `pthread_cond_wait()`, which can cause an infinite wait.

pthread_cond_broadcast Return Values

`pthread_cond_broadcast()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: *cv* points to an illegal address.

Destroying the Condition Variable State

Use `pthread_cond_destroy(3C)` to destroy any state that is associated with the condition variable pointed to by *cv*.

pthread_cond_destroy Syntax

```
int pthread_cond_destroy(pthread_cond_t *cv);

#include <pthread.h>

pthread_cond_t cv;
int ret;

/* Condition variable is destroyed */
ret = pthread_cond_destroy(&cv);
```

Note that the space for storing the condition variable is not freed.

pthread_cond_destroy Return Values

`pthread_cond_destroy()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value specified by *cv* is invalid.

Lost Wake-Up Problem

A call to `pthread_cond_signal()` or `pthread_cond_broadcast()` when the thread does not hold the mutex lock associated with the condition can lead to *lost wake-up* bugs.

A lost wake-up occurs when all of the following conditions are in effect:

- A thread calls `pthread_cond_signal()` or `pthread_cond_broadcast()`
- Another thread is between the test of the condition and the call to `pthread_cond_wait()`
- No threads are waiting

  The signal has no effect, and therefore is lost

This can occur only if the condition being tested is modified without holding the mutex lock associated with the condition. As long as the condition being tested is modified only while holding the associated mutex, `pthread_cond_signal()` and `pthread_cond_broadcast()` can be called regardless of whether they are holding the associated mutex.

Producer and Consumer Problem

The producer and consumer problem is one of the small collection of standard, well-known problems in concurrent programming. A finite-size buffer and two classes of threads, producers and consumers, put items into the buffer (producers) and take items out of the buffer (consumers).

A producer cannot put something in the buffer until the buffer has space available. A consumer cannot take something out of the buffer until the producer has written to the buffer.

A condition variable represents a queue of threads that wait for some condition to be signaled.

Example 4–11 has two such queues. One (*less*) queue for producers waits for a slot in the buffer. The other (*more*) queue for consumers waits for a buffer slot containing information. The example also has a mutex, as the data structure describing the buffer must be accessed by only one thread at a time.

EXAMPLE 4–11 Producer and Consumer Problem With Condition Variables

```
typedef struct {
     char buf[BSIZE];
     int occupied;
     int nextin;
     int nextout;
     pthread_mutex_t mutex;
     pthread_cond_t more;
     pthread_cond_t less; }
buffer_t;

buffer_t buffer;
```

As Example 4–12 shows, the producer thread acquires the mutex protecting the `buffer` data structure. The producer thread then makes certain that space is available for the item produced. If space is not available, the producer thread calls `pthread_cond_wait()`. `pthread_cond_wait()` causes the producer thread to join the queue of threads that are waiting for the condition *less* to be signaled. *less* represents available room in the buffer.

At the same time, as part of the call to `pthread_cond_wait()`, the thread releases its lock on the mutex. The waiting producer threads depend on consumer threads to signal when the condition is true, as shown in Example 4–12. When the condition is signaled, the first thread waiting on *less* is awakened. However, before the thread can return from `pthread_cond_wait()`, the thread must acquire the lock on the mutex again.

Acquire the mutex to ensure that the thread again has mutually exclusive access to the buffer data structure. The thread then must check that available room in the buffer actually exists. If room is available, the thread writes into the next available slot.

At the same time, consumer threads might be waiting for items to appear in the buffer. These threads are waiting on the condition variable *more* . A producer thread, having just deposited something in the buffer, calls `pthread_cond_signal()` to wake up the next waiting consumer. If no consumers are waiting, this call has no effect.

Finally, the producer thread unlocks the mutex, allowing other threads to operate on the buffer data structure.

EXAMPLE 4–12 The Producer and Consumer Problem: the Producer

```
void producer(buffer_t *b, char item)
{
    pthread_mutex_lock(&b->mutex);

    while (b->occupied >= BSIZE)
        pthread_cond_wait(&b->less, &b->mutex);

    assert(b->occupied < BSIZE);

    b->buf[b->nextin++] = item;

    b->nextin %= BSIZE;
    b->occupied++;

    /* now: either b->occupied < BSIZE and b->nextin is the index
       of the next empty slot in the buffer, or
       b->occupied == BSIZE and b->nextin is the index of the
       next (occupied) slot that will be emptied by a consumer
       (such as b->nextin == b->nextout) */

    pthread_cond_signal(&b->more);

    pthread_mutex_unlock(&b->mutex);
}
```

Note the use of the `assert()` statement. Unless the code is compiled with `NDEBUG` defined, `assert()` does nothing when its argument evaluates to true (nonzero). The program aborts if the argument evaluates to false (zero). Such assertions are especially useful in multithreaded programs. `assert()` immediately points out runtime problems if the assertion fails. `assert()` has the additional effect of providing useful comments.

The comment that begins `/* now: either b->occupied ...` could better be expressed as an assertion, but the statement is too complicated as a Boolean-valued expression and so is given in English.

Both assertions and comments are examples of invariants. These invariants are logical statements that should not be falsified by the execution of the program with the following exception. The exception occurs during brief moments when a thread is modifying some of the program variables mentioned in the invariant. An assertion, of course, should be true whenever any thread executes the statement.

The use of invariants is an extremely useful technique. Even if the invariants are not stated in the program text, think in terms of invariants when you analyze a program.

The invariant in the producer code that is expressed as a comment is always true whenever a thread executes the code where the comment appears. If you move this comment to just after the `mutex_unlock()`, the comment does not necessarily remain true. If you move this comment to just after the `assert()`, the comment is still true.

This invariant therefore expresses a property that is true at all times with the following exception. The exception occurs when either a producer or a consumer is changing the state of the buffer. While a thread is operating on the buffer under the protection of a mutex, the thread might temporarily falsify the invariant. However, once the thread is finished, the invariant should be true again.

Example 4–13 shows the code for the consumer. The logic flow is symmetric with the logic flow of the producer.

EXAMPLE 4–13 The Producer and Consumer Problem: the Consumer

```
char consumer(buffer_t *b)
{
    char item;
    pthread_mutex_lock(&b->mutex);
    while(b->occupied <= 0)
        pthread_cond_wait(&b->more, &b->mutex);

    assert(b->occupied > 0);

    item = b->buf[b->nextout++];
    b->nextout %= BSIZE;
    b->occupied--;

    /* now: either b->occupied > 0 and b->nextout is the index
       of the next occupied slot in the buffer, or
       b->occupied == 0 and b->nextout is the index of the next
       (empty) slot that will be filled by a producer (such as
       b->nextout == b->nextin) */

    pthread_cond_signal(&b->less);
    pthread_mutex_unlock(&b->mutex);

    return(item);
```

EXAMPLE 4–13 The Producer and Consumer Problem: the Consumer *(Continued)*

```
}
```

Synchronization With Semaphores

A semaphore is a programming construct designed by E. W. Dijkstra in the late 1960s. Dijkstra's model was the operation of railroads. Consider a stretch of railroad where a single track is present over which only one train at a time is allowed.

A semaphore synchronizes travel on this track. A train must wait before entering the single track until the semaphore is in a state that permits travel. When the train enters the track, the semaphore changes state to prevent other trains from entering the track. A train that is leaving this section of track must again change the state of the semaphore to allow another train to enter.

In the computer version, a semaphore appears to be a simple integer. A thread waits for permission to proceed and then signals that the thread has proceeded by performing a `P` operation on the semaphore.

The thread must wait until the semaphore's value is positive, then change the semaphore's value by subtracting 1 from the value. When this operation is finished, the thread performs a `V` operation, which changes the semaphore's value by adding 1 to the value. These operations must take place atomically. These operations cannot be subdivided into pieces between which other actions on the semaphore can take place. In the `P` operation, the semaphore's value must be positive just before the value is decremented, resulting in a value that is guaranteed to be nonnegative and 1 less than what it was before it was decremented.

In both `P` and `V` operations, the arithmetic must take place without interference. The net effect of two `V` operations performed simultaneously on the same semaphore, should be that the semaphore's new value is 2 greater than it was.

The mnemonic significance of `P` and `V` is unclear to most of the world, as Dijkstra is Dutch. However, in the interest of true scholarship: `P` stands for prolagen, a made-up word derived from proberen te verlagen, which means *try to decrease*. `V` stands for verhogen, which means *increase*. The mnemonic significance is discussed in one of Dijkstra's technical notes, EWD 74.

`sem_wait`(3RT) and `sem_post`(3RT) correspond to Dijkstra's `P` and `V` operations. `sem_trywait`(3RT) is a conditional form of the `P` operation. If the calling thread cannot decrement the value of the semaphore without waiting, the call to returns immediately with a nonzero value.

The two basic sorts of semaphores are binary semaphores and counting semaphores. Binary semaphores never take on values other than zero or one, and counting semaphores take on arbitrary nonnegative values. A binary semaphore is logically just like a mutex.

However, although not always enforced, mutexes should be unlocked only by the thread that holds the lock. Because no notion exists of "the thread that holds the semaphore," any thread can perform a v or sem_post (3RT) operation.

Counting semaphores are nearly as powerful as conditional variables when used in conjunction with mutexes. In many cases, the code might be simpler when implemented with counting semaphores rather than with condition variables, as shown in Example 4–14, Example 4–15, and Example 4–16.

However, when a mutex is used with condition variables, an implied bracketing is present. The bracketing clearly delineates which part of the program is being protected. This behavior is not necessarily the case for a semaphore, which might be called the *go to* of concurrent programming. A semaphore is powerful but too easy to use in an unstructured, indeterminate way.

Named and Unnamed Semaphores

POSIX semaphores can be unnamed or named. Unnamed semaphores are allocated in process memory and initialized. Unnamed semaphores might be usable by more than one process, depending on how the semaphore is allocated and initialized. Unnamed semaphores are either private, inherited through fork(), or are protected by access protections of the regular file in which they are allocated and mapped.

Named semaphores are like process-shared semaphores, except that named semaphores are referenced with a pathname rather than a *pshared* value. Named semaphores are sharable by several processes. Named semaphores have an owner user-id, group-id, and a protection mode.

The functions sem_open, sem_getvalue, sem_close, and sem_unlink are available to open, retrieve, close, and remove named semaphores. By using sem_open, you can create a named semaphore that has a name defined in the file system name space.

For more information about named semaphores, see the sem_open, sem_getvalue, sem_close, and sem_unlink man pages.

Counting Semaphores Overview

Conceptually, a semaphore is a nonnegative integer count. Semaphores are typically used to coordinate access to resources, with the semaphore count initialized to the number of free resources. Threads then atomically increment the count when resources are added and atomically decrement the count when resources are removed.

When the semaphore count becomes zero, no more resources are present. Threads that try to decrement the semaphore when the count is zero block until the count becomes greater than zero.

TABLE 4–6 Routines for Semaphores

Operation	Related Function Description
Initialize a semaphore	"sem_init Syntax" on page 121
Increment a semaphore	"sem_post Syntax" on page 123
Block on a semaphore count	"sem_wait Syntax" on page 123
Decrement a semaphore count	"sem_trywait Syntax" on page 124
Destroy the semaphore state	"sem_destroy Syntax" on page 124

Because semaphores need not be acquired and be released by the same thread, semaphores can be used for asynchronous event notification, such as in signal handlers. And, because semaphores contain state, semaphores can be used asynchronously without acquiring a mutex lock as is required by condition variables. However, semaphores are not as efficient as mutex locks.

The scheduling policy determines the order in which blocked threads are awakened. The default scheduling policy, `SCHED_OTHER`, does not specify the order in which threads are awakened. Under the `SCHED_FIFO` and `SCHED_RR` real-time scheduling policies, threads are awakened in priority order.

Semaphores must be initialized before use, however semaphores do not have attributes.

Initializing a Semaphore

Use `sem-init` to initialize the unnamed semaphore variable pointed to by *sem* to *value* amount.

sem_init Syntax

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

```
#include <semaphore.h>

sem_t sem;
int pshared;
int ret;
int value;

/* initialize a private semaphore */
pshared = 0;
value = 1;
ret = sem_init(&sem, pshared, value);
```

If the value of *pshared* is zero, then the semaphore cannot be shared between processes. If the value of *pshared* is nonzero, then the semaphore can be shared between processes.

Multiple threads must not initialize the same semaphore.

A semaphore must not be reinitialized while other threads might be using the semaphore.

Initializing Semaphores With Intraprocess Scope

When *pshared* is 0, the semaphore can be used by all the threads in this process only.

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* to be used within this process only */
ret = sem_init(&sem, 0, count);
```

Initializing Semaphores With Interprocess Scope

When *pshared* is nonzero, the semaphore can be shared by other processes.

```
#include <semaphore.h>

sem_t sem;
int ret;
int count = 4;

/* to be shared among processes */
ret = sem_init(&sem, 1, count);
```

sem_init Return Values

`sem_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value argument exceeds `SEM_VALUE_MAX`.

`ENOSPC`

Description: A resource that is required to initialize the semaphore has been exhausted. The limit on semaphores `SEM_NSEMS_MAX` has been reached.

`EPERM`

Description: The process lacks the appropriate privileges to initialize the semaphore.

Incrementing a Semaphore

Use `sem_post` to atomically increment the semaphore pointed to by *sem*.

sem_post Syntax

```
int sem_post(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_post(&sem); /* semaphore is posted */
```

When any threads are blocked on the semaphore, one of the threads is unblocked.

sem_post Return Values

`sem_post()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: *sem* points to an illegal address.

Blocking on a Semaphore Count

Use `sem_wait` to block the calling thread until the semaphore count pointed to by *sem* becomes greater than zero, then atomically decrement the count.

sem_wait Syntax

```
int sem_wait(sem_t *sem);
```

```
#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_wait(&sem); /* wait for semaphore */
```

sem_wait Return Values

`sem_wait()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: *sem* points to an illegal address.

`EINTR`

Description: A signal interrupted this function.

Decrementing a Semaphore Count

Use `sem_trywait` to try to atomically decrement the count in the semaphore pointed to by *sem* when the count is greater than zero.

sem_trywait Syntax

```
int sem_trywait(sem_t *sem);

#include <semaphore.h>

sem_t sem;
int ret;

ret = sem_trywait(&sem); /* try to wait for semaphore*/
```

This function is a nonblocking version of `sem_wait()`. `sem_trywait()` returns immediately if unsuccessful.

sem_trywait Return Values

`sem_trywait()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: *sem* points to an illegal address.

`EINTR`

Description: A signal interrupted this function.

`EAGAIN`

Description: The semaphore was already locked, so the semaphore cannot be immediately locked by the `sem_trywait()` operation.

Destroying the Semaphore State

Use `sem_destroy` to destroy any state that is associated with the unnamed semaphore pointed to by *sem*.

sem_destroy Syntax

```
int sem_destroy(sem_t *sem);

#include <semaphore.h>

sem_t sem;
```

```
int ret;

ret = sem_destroy(&sem); /* the semaphore is destroyed */
```

The space for storing the semaphore is not freed.

sem_destroy Return Values

`sem_destroy()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: *sem* points to an illegal address.

Producer and Consumer Problem Using Semaphores

The data structure in Example 4–14 is similar to the structure used for the condition variables example, shown in Example 4–11. Two semaphores represent the number of full and empty buffers. The semaphores ensure that producers wait until buffers are empty and that consumers wait until buffers are full.

EXAMPLE 4–14 Producer and Consumer Problem With Semaphores

```
typedef struct {
    char buf[BSIZE];
    sem_t occupied;
    sem_t empty;
    int nextin;
    int nextout;
    sem_t pmut;
    sem_t cmut;
} buffer_t;

buffer_t buffer;

sem_init(&buffer.occupied, 0, 0);
sem_init(&buffer.empty,0, BSIZE);
sem_init(&buffer.pmut, 0, 1);
sem_init(&buffer.cmut, 0, 1);
buffer.nextin = buffer.nextout = 0;
```

Another pair of binary semaphores plays the same role as mutexes. The semaphores control access to the buffer when multiple producers use multiple empty buffer slots, and when multiple consumers use multiple full buffer slots. Mutexes would work better here, but would not provide as good an example of semaphore use.

EXAMPLE 4–15 Producer and Consumer Problem: the Producer

```
void producer(buffer_t *b, char item) {
    sem_wait(&b->empty);
    sem_wait(&b->pmut);

    b->buf[b->nextin] = item;
    b->nextin++;
    b->nextin %= BSIZE;

    sem_post(&b->pmut);
    sem_post(&b->occupied);
}
```

EXAMPLE 4–16 Producer and Consumer Problem: the Consumer

```
char consumer(buffer_t *b) {
    char item;

    sem_wait(&b->occupied);

    sem_wait(&b->cmut);

    item = b->buf[b->nextout];
    b->nextout++;
    b->nextout %= BSIZE;

    sem_post(&b->cmut);

    sem_post(&b->empty);

    return(item);
}
```

Read-Write Lock Attributes

Read-write locks permit concurrent reads and exclusive writes to a protected shared resource. The read-write lock is a single entity that can be locked in *read* or *write* mode. To modify a resource, a thread must first acquire the exclusive write lock. An exclusive write lock is not permitted until all read locks have been released.

Database access can be synchronized with a read-write lock. Read-write locks support concurrent reads of database records because the read operation does not change the record's information. When the database is to be updated, the write operation must acquire an exclusive write lock.

To change the default read-write lock attributes, you can declare and initialize an attribute object. Often, the read-write lock attributes are set up in one place at the beginning of the application. Set up at the beginning of the application makes the attributes easier to locate and modify. The following table lists the functions discussed in this section that manipulate read-write lock attributes.

TABLE 4–7 Routines for Read-Write Lock Attributes

Operation	Related Function Description
Initialize a read-write lock attribute	"pthread_rwlockattr_init Syntax" on page 127
Destroy a read-write lock attribute	"pthread_rwlockattr_destroy Syntax" on page 127
Set a read-write lock attribute	"pthread_rwlockattr_setpshared Syntax" on page 128
Get a read-write lock attribute	"pthread_rwlockattr_getpshared Syntax" on page 129

Initializing a Read-Write Lock Attribute

pthread_rwlockattr_init(3C) initializes a read-write lock attributes object *attr* with the default value for all of the attributes defined by the implementation.

pthread_rwlockattr_init Syntax

```
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

Results are undefined if pthread_rwlockattr_init is called specifying an already initialized read-write lock attributes object. After a read-write lock attributes object initializes one or more read-write locks, any function that affects the object, including destruction, does not affect previously initialized read-write locks.

pthread_rwlockattr_init Return Values

If successful, pthread_rwlockattr_init() returns zero. Otherwise, an error number is returned to indicate the error.

ENOMEM

Description: Insufficient memory exists to initialize the read-write attributes object.

Destroying a Read-Write Lock Attribute

pthread_rwlockattr_destroy(3C) destroys a read-write lock attributes object.

pthread_rwlockattr_destroy Syntax

```
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

The effect of subsequent use of the object is undefined until the object is re-initialized by another call to pthread_rwlockattr_init(). An implementation can cause pthread_rwlockattr_destroy() to set the object referenced by *attr* to an invalid value.

pthread_rwlockattr_destroy Return Values

If successful, `pthread_rwlockattr_destroy()` returns zero. Otherwise, an error number is returned to indicate the error.

`EINVAL`
Description: The value specified by *attr* is invalid.

Setting a Read-Write Lock Attribute

`pthread_rwlockattr_setpshared`(3C) sets the process-shared read-write lock attribute.

pthread_rwlockattr_setpshared Syntax

```
#include <pthread.h>
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

The *pshared* lock attribute has one of the following values:

`PTHREAD_PROCESS_SHARED`
Description: Permits a read-write lock to be operated on by any thread that has access to the memory where the read-write lock is allocated. Operation on the read-write lock is permitted even if the lock is allocated in memory that is shared by multiple processes.

`PTHREAD_PROCESS_PRIVATE`
Description: The read-write lock is only operated upon by threads created within the same process as the thread that initialized the read-write lock. If threads of differing processes attempt to operate on such a read-write lock, the behavior is undefined. The default value of the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`.

pthread_rwlockattr_setpshared Return Values

If successful, `pthread_rwlockattr_setpshared()` returns zero. Otherwise, an error number is returned to indicate the error.

`EINVAL`
Description: The value specified by *attr* or *pshared* is invalid.

Getting a Read-Write Lock Attribute

`pthread_rwlockattr_getpshared`(3C) gets the process-shared read-write lock attribute.

pthread_rwlockattr_getpshared Syntax

```
#include <pthread.h>
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
          int *restrict pshared);
```

`pthread_rwlockattr_getpshared()` obtains the value of the process-shared attribute from the initialized attributes object referenced by *attr*.

pthread_rwlockattr_getpshared Return Values

If successful, `pthread_rwlockattr_getpshared()` returns zero. Otherwise, an error number is returned to indicate the error.

`EINVAL`

Description: The value specified by *attr* or *pshared* is invalid.

Using Read-Write Locks

After the attributes for a read-write lock are configured, you initialize the read-write lock. The following functions are used to initialize or destroy, lock or unlock, or try to lock a read-write lock. The following table lists the functions discussed in this section that manipulate read-write locks.

TABLE 4–8 Routines that Manipulate Read-Write Locks

Operation	Related Function Description
Initialize a read-write lock	"`pthread_rwlock_init` Syntax" on page 130
Read lock on read-write lock	"`pthread_rwlock_rdlock` Syntax" on page 130
Read lock with a nonblocking read-write lock	"`pthread_rwlock_tryrdlock` Syntax" on page 132
Write lock on read-write lock	"`pthread_rwlock_wrlock` Syntax" on page 133
Write lock with a nonblocking read-write lock	"`pthread_rwlock_trywrlock` Syntax" on page 134
Unlock a read-write lock	"`pthread_rwlock_unlock` Syntax" on page 135
Destroy a read-write lock	"`pthread_rwlock_destroy` Syntax" on page 136

Initializing a Read-Write Lock

Use `pthread_rwlock_init`(3C) to initialize the read-write lock referenced by *rwlock* with the attributes referenced by *attr*.

pthread_rwlock_init Syntax

```
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
          const pthread_rwlockattr_t *restrict attr);

pthread_rwlock_t  rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

If *attr* is `NULL`, the default read-write lock attributes are used. The effect is the same as passing the address of a default read-write lock attributes object. After the lock is initialized, the lock can be used any number of times without being re-initialized. On successful initialization, the state of the read-write lock becomes initialized and unlocked. Results are undefined if `pthread_rwlock_init()` is called specifying an already initialized read-write lock. Results are undefined if a read-write lock is used without first being initialized.

In cases where default read-write lock attributes are appropriate, the macro `PTHREAD_RWLOCK_INITIALIZER` can initialize read-write locks that are statically allocated. The effect is equivalent to dynamic initialization by a call to `pthread_rwlock_init()` with the parameter *attr* specified as `NULL`, except that no error checks are performed.

pthread_rwlock_init Return Values

If successful, `pthread_rwlock_init()` returns zero. Otherwise, an error number is returned to indicate the error.

If `pthread_rwlock_init()` fails, *rwlock* is not initialized and the contents of *rwlock* are undefined.

`EINVAL`

Description: The value specified by *attr* or *rwlock* is invalid.

Acquiring the Read Lock on Read-Write Lock

`pthread_rwlock_rdlock`(3C) applies a read lock to the read-write lock referenced by *rwlock*.

pthread_rwlock_rdlock Syntax

```
#include <pthread.h>

int  pthread_rwlock_rdlock(pthread_rwlock_t *rwlock );
```

The calling thread acquires the read lock if a writer does not hold the lock and no writers are blocked on the lock. Whether the calling thread acquires the lock when a writer does not hold the lock and writers are waiting for the lock is unspecified. If a writer holds the lock, the calling thread does not acquire the read lock. If the read lock is not acquired, the calling thread blocks. The thread does not return from the `pthread_rwlock_rdlock()` until the thread can acquire the lock. Results are undefined if the calling thread holds a write lock on *rwlock* at the time the call is made.

Implementations are allowed to favor writers over readers to avoid writer starvation. The Oracle Solaris implementation favors writers over readers.

A thread can hold multiple concurrent read locks on *rwlock* The thread can successfully call `pthread_rwlock_rdlock()` *n* times. The thread must call `pthread_rwlock_unlock()` *n* times to perform matching unlocks.

Results are undefined if `pthread_rwlock_rdlock()` is called with an uninitialized read-write lock.

A thread signal handler processes a signal delivered to a thread waiting for a read-write lock. On return from the signal handler, the thread resumes waiting for the read-write lock for reading as if the thread was not interrupted.

pthread_rwlock_rdlock Return Values

If successful, `pthread_rwlock_rdlock()` returns zero. Otherwise, an error number is returned to indicate the error.

`EINVAL`

Description: The value specified by *attr* or *rwlock* is invalid.

Acquiring a Read Lock on a Read-Write Lock Before a Specified Absolute Time

The `pthread_rwlock_timedrdlock(3C)` function applies a read lock to the read-write lock referenced by *rwlock* as in the `pthread_rwlock_rdlock()` function.

pthread_rwlock_timedrdlock Syntax

```
#include <pthread.h>
#include <time.h>

int  pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
          const struct timespec *restrict abs_timeout);
```

If the lock cannot be acquired without waiting for other threads to unlock the lock, this wait will be terminated when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the `CLOCK_REALTIME` clock (that is, when the value of that clock equals or exceeds *abs_timeout*), or if the absolute time specified by *abs_timeout* has already been passed at the time of the call.

The resolution of the timeout is the resolution of the `CLOCK_REALTIME` clock. The `timespec` data type is defined in the `<time.h>` header. Under no circumstances does the function fail with a timeout if the lock can be acquired immediately. The validity of the timeout parameter need not be checked if the lock can be immediately acquired.

If a signal that causes a signal handler to be executed is delivered to a thread blocked on a read-write lock with a call to `pthread_rwlock_timedrdlock()`, upon return from the signal handler the thread resumes waiting for the lock as if it was not interrupted.

The calling thread might deadlock if at the time the call is made it holds a write lock on *rwlock*.

The `pthread_rwlock_reltimedrdlock_np()` function is identical to the `pthread_rwlock_timedrdlock()` function, except that the timeout is specified as a relative time interval.

pthread_rwlock_timedrdlock Return Values

If successful, returns 0 if the lock for writing on the read-write lock object referenced by rwlock is acquired. Otherwise, an error number is returned to indicate the error.

`ETIMEDOUT`

Description: The lock could not be acquired before the specified timeout expired.

`EAGAIN`

Description: The read lock could not be acquired because the maximum number of read locks for lock would be exceeded.

`EDEADLK`

Description: The calling thread already holds the *rwlock*.

`EINVAL`

Description: The value specified by *rwlock* does not refer to an initialized read-write lock object, or the timeout nanosecond value is less than zero or greater than or equal to 1,000 million.

Acquiring a Non-Blocking Read Lock on a Read-Write Lock

`pthread_rwlock_tryrdlock`(3C) applies a read lock as in `pthread_rwlock_rdlock()` with the exception that the function fails if any thread holds a write lock on *rwlock* or writers are blocked on *rwlock*.

pthread_rwlock_tryrdlock Syntax

```
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

pthread_rwlock_tryrdlock Return Values

`pthread_rwlock_tryrdlock()` returns zero if the lock for reading on the read-write lock object referenced by *rwlock* is acquired. If the lock is not acquired, an error number is returned to indicate the error.

`EBUSY`

Description: The read-write lock could not be acquired for reading because a writer holds the lock or was blocked on it.

Acquiring the Write Lock on a Read-Write Lock

`pthread_rwlock_wrlock`(3C) applies a write lock to the read-write lock referenced by *rwlock*.

pthread_rwlock_wrlock Syntax

```
#include <pthread.h>

int  pthread_rwlock_wrlock(pthread_rwlock_t *rwlock );
```

The calling thread acquires the write lock if no other reader thread or writer thread holds the read-write lock *rwlock*. Otherwise, the thread blocks. The thread does not return from the `pthread_rwlock_wrlock()` call until the thread can acquire the lock. Results are undefined if the calling thread holds the read-write lock, either a read lock or write lock, at the time the call is made.

Implementations are allowed to favor writers over readers to avoid writer starvation. The Oracle Solaris implementation favors writers over readers.

Results are undefined if `pthread_rwlock_wrlock()` is called with an uninitialized read-write lock.

The thread signal handler processes a signal delivered to a thread waiting for a read-write lock for writing. Upon return from the signal handler, the thread resumes waiting for the read-write lock for writing as if the thread was not interrupted.

pthread_rwlock_wrlock Return Values

`pthread_rwlock_rwlock()` returns zero if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. If the lock is not acquired, an error number is returned to indicate the error.

Acquiring a Non-blocking Write Lock on a Read-Write Lock

pthread_rwlock_trywrlock(3C) applies a write lock like pthread_rwlock_wrlock(), with the exception that the function fails if any thread currently holds *rwlock*, for reading or writing.

pthread_rwlock_trywrlock Syntax

```
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t  *rwlock);
```

Results are undefined if pthread_rwlock_trywrlock() is called with an uninitialized read-write lock.

pthread_rwlock_trywrlock Return Values

If successful, pthread_rwlock_trywrlock() returns zero if the lock for writing on the read-write lock object referenced by *rwlock* is acquired. Otherwise, an error number is returned to indicate the error.

EBUSY

Description: The read-write lock could not be acquired for writing because the read-write lock is already locked for reading or writing.

Acquiring a Write Lock on a Read-Write Lock Before a Specified Absolute Time

The pthread_rwlock_timedwrlock(3C) function applies a write lock to the read-write lock referenced by *rwlock* as in the pthread_rwlock_wrlock() function, but attempts to apply the lock only until a specified absolute time.

pthread_rwlock_timedwrlock Syntax

```
#include <pthread.h>
#include <time.h>

int  pthread_rwlock_timedwrlock(pthread_rwlock_t   *restrict rwlock,
     const struct timespec *restrict abs_timeout);
```

The calling thread acquires the write lock if no other reader thread or writer thread holds the read-write lock *rwlock*. If the lock cannot be acquired without waiting for other threads to unlock the lock, this wait will be terminated when the specified timeout expires. The timeout expires when the absolute time specified by *abs_timeout*passes, as measured by the CLOCK_REALTIME clock (that is, when the value of that clock equals or exceeds *abs_timeout*) or if the absolute time specified by *abs_timeout* has already been passed at the time of the call. The `pthread_rwlock_reltimedwrlock_np()` function is identical to the `pthread_rwlock_timedwrlock()` function, except that the timeout is specified as a relative time interval.

pthread_rwlock_timedwrlock Returns

If successful, returns 0 if the lock for writing on the read-write lock object referenced by rwlock is acquired. Otherwise, an error number is returned to indicate the error.

`ETIMEDOUT`

Description: The lock could not be acquired before the specified timeout expired.

`EDEADLK`

Description: The calling thread already holds the *rwlock*.

`EINVAL`

Description: The value specified by *rwlock* does not refer to an initialized read-write lock object, or the timeout nanosecond value is less than zero or greater than or equal to 1,000 million.

Unlocking a Read-Write Lock

`pthread_rwlock_unlock`(3C) releases a lock held on the read-write lock object referenced by *rwlock*.

pthread_rwlock_unlock Syntax

```
#include <pthread.h>

int pthread_rwlock_unlock (pthread_rwlock_t  *rwlock);
```

Results are undefined if the read-write lock *rwlock* is not held by the calling thread.

If `pthread_rwlock_unlock()` is called to release a read lock from the read-write lock object, and other read locks are currently held on this lock object, the object remains in the read locked state. If `pthread_rwlock_unlock()` releases the calling thread's last read lock on this read-write lock object, the calling thread is no longer an owner of the object. If `pthread_rwlock_unlock()` releases the last read lock for this read-write lock object, the read-write lock object is put in the unlocked state with no owners.

If `pthread_rwlock_unlock()` is called to release a write lock for this read-write lock object, the lock object is put in the unlocked state with no owners.

If `pthread_rwlock_unlock()` unlocks the read-write lock object and multiple threads are waiting to acquire the lock object for writing, the scheduling policy determines which thread acquires the object for writing. If multiple threads are waiting to acquire the read-write lock object for reading, the scheduling policy determines the order the waiting threads acquire the object for reading. If multiple threads are blocked on *rwlock* for both read locks and write locks, whether the readers or the writer acquire the lock first is unspecified.

Results are undefined if `pthread_rwlock_unlock()` is called with an uninitialized read-write lock.

pthread_rwlock_unlock Return Values

If successful, `pthread_rwlock_unlock()` returns zero. Otherwise, an error number is returned to indicate the error.

Destroying a Read-Write Lock

`pthread_rwlock_destroy`(3C) destroys the read-write lock object referenced by *rwlock* and releases any resources used by the lock.

pthread_rwlock_destroy Syntax

```
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t **rwlock);
```

The effect of subsequent use of the lock is undefined until the lock is re-initialized by another call to `pthread_rwlock_init()`. An implementation can cause `pthread_rwlock_destroy()` to set the object referenced by *rwlock* to an invalid value. Results are undefined if `pthread_rwlock_destroy()` is called when any thread holds *rwlock*. Attempting to destroy an uninitialized read-write lock results in undefined behavior. A destroyed read-write lock object can be re-initialized using `pthread_rwlock_init()`. The results of otherwise referencing the read-write lock object after the lock object has been destroyed are undefined.

pthread_rwlock_destroy Return Values

If successful, `pthread_rwlock_destroy()` returns zero. Otherwise, an error number is returned to indicate the error.

`EINVAL`

Description: The value specified by *attr* or *rwlock* is invalid.

Using Barrier Synchronization

In cases where you must wait for a number of tasks to be completed before an overall task can proceed, *barrier synchronization* can be used. POSIX threads specifies a synchronization object called a *barrier*, along with barrier functions. The functions create the barrier, specifying the number of threads that are synchronizing on the barrier, and set up threads to perform tasks and wait at the barrier until all the threads reach the barrier. When the last thread arrives at the barrier, all the threads resume execution.

See "Parallelizing a Loop on a Shared-Memory Parallel Computer" on page 233 for more about barrier synchronization.

Initializing a Synchronization Barrier

Use `pthread_barrier_init(3C)` to allocate resources for a barrier and initialize its attributes.

`pthread_barrier_init()` Syntax

```
int pthread_barrier_init(pthread_barrier_t  *barrier,
          const pthread_barrierattr_t *restrict attr,
          unsigned count);
```

```
#include <pthread.h>
pthread_barrier_t barrier;
pthread_barrierattr_t attr;
unsigned count;
int ret;
ret = pthread_barrier_init(&barrier, &attr, count);
```

The `pthread_barrier_init()` function allocates any resources required to use the barrier referenced by *barrier* and initializes the barrier with attributes referenced by *attr*. If *attr* is NULL, the default barrier attributes are used; the effect is the same as passing the address of a default barrier attributes object. The *count* argument specifies the number of threads that must call `pthread_barrier_wait()` before any of them successfully return from the call. The value specified by *count* must be greater than 0.

`pthread_barrier_init()` Return Values

`pthread_barrier_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: The value specified by *count* is equal to 0, or the value specified by *attr* is invalid

EAGAIN

Description: The system lacks the necessary resources to initialize another barrier.

ENOMEM

Description: Insufficient memory exists to initialize the barrier.

EBUSY

Description: There was an attempt to destroy a barrier while it is in use (for example, while being used in a `pthread_barrier_wait()` call) by another thread.

Waiting for Threads to Synchronize at a Barrier

Use `pthread_barrier_wait(3C)` to synchronize threads at a specified barrier. The calling thread blocks until the required number of threads have called `pthread_barrier_wait()` specifying the barrier. The number of threads is specified in the `pthread_barrier_init()` function.

When the required number of threads have called `pthread_barrier_wait()` specifying the barrier, the constant `PTHREAD_BARRIER_SERIAL_THREAD` is returned to one unspecified thread and 0 is returned to each of the remaining threads. The barrier is then reset to the state it had as a result of the most recent `pthread_barrier_init()` function that referenced it.

`pthread_barrier_wait()` Syntax

```
int pthread_barrier_wait(pthread_barrier_t  *barrier);
```

```
#include <pthread.h>
pthread_barrier_t barrier;
int ret;
ret = pthread_barrier_wait(&barrier);
```

`pthread_barrier_wait()` Return Values

When `pthread_barrier_wait()` completes successfully, the function returns `PTHREAD_BARRIER_SERIAL_THREAD`, which is defined in `pthread.h`, for one arbitrary thread synchronized at the barrier. The function returns zero for each of the other threads. Otherwise an error code is returned.

EINVAL

Description: The value specified by *barrier* does not refer to an initialized barrier object.

Destroying a Synchronization Barrier

When a barrier is no longer needed, it should be destroyed. Use the `pthread_barrier_destroy`(3C) function to destroy the barrier referenced by *barrier* and release any resources used by the barrier.

pthread_barrier_destroy Syntax

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

```
#include <pthread.h>
pthread_barrier_t barrier;
int ret;
ret = pthread_barrier_destroy(&barrier);
```

pthread_barrier_destroy Return Values

`pthread_barrier_destroy()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Indicates that the value of *barrier* was not valid.

`EBUSY`

Description: An attempt was made to destroy a barrier while it is in use (for example, while being used in a `pthread_barrier_wait()` by another thread.

Initializing a Barrier Attributes Object

The `pthread_barrierattr_init`(3C) function initializes a barrier attributes object *attr* with the default values for the attributes defined for the object by the implementation. Currently, only the process-shared attribute is provided, and the `pthread_barrierattr_getpshared()` and `pthread_barrierattr_setpshared()` functions are used to get and set the attribute.

After a barrier attributes object has been used to initialize one or more barriers, any function affecting the attributes object (including destruction) does not affect any previously initialized barrier.

`pthread_barrierattr_init()` Syntax

```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

```
#include <pthread.h>
pthread_barrierattr_t attr;
int ret;
ret = pthread_barrierattr_init(&attr);
```

`pthread_barrierattr_init()` Return Values

`pthread_barrierattr_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`ENOMEM`

Description: Insufficient memory exists to initialize the barrier attributes object.

Setting a Barrier Process-Shared Attribute

The `pthread_barrierattr_setpshared()` function sets the process-shared attribute in an initialized attributes object referenced by *attr*. The process-shared attribute can have the following values:

`PTHREAD_PROCESS_PRIVATE`	The barrier can only be operated upon by threads created within the same process as the thread that initialized the barrier. This is the default value of the process-shared attribute.
`PTHREAD_PROCESS_SHARED`	The barrier can be operated upon by any thread that has access to the memory where the barrier is allocated.

`pthread_barrierattr_setpshared()` Syntax

```
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);
```

`pthread_barrierattr_setpshared()` Return Values

`pthread_barrierattr_setpshared()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Indicates that the value of *attr* was not valid, or the new value specified for the *pshared* is not valid.

Getting a Barrier Process-Shared Attribute

The `pthread_barrierattr_getpshared`(3C) function obtains the value of the process-shared attribute from the attributes object referenced by *attr*. The value is set by the `pthread_barrierattr_setpshared()` function.

`pthread_barrierattr_getpshared()` Syntax

```
int pthread_barrierattr_getpshared(const pthread_barrierattr_t *restrict attr,
          int *restrict pshared);
```

`pthread_barrierattr_getpshared()` Return Values

`pthread_barrierattr_getpshared()` returns zero after completing successfully, and stores the value of the process-shared attribute of *attr* into the object referenced by the *pshared* parameter. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Indicates that the value of *attr* was not valid.

Destroying a Barrier Attributes Object

The `pthread_barrierattr_destroy()` function destroys a barrier attributes object. A destroyed *attr* attributes object can be reinitialized using `pthread_barrierattr_init()`.

After a barrier attributes object has been used to initialize one or more barriers, destroying the object does not affect any previously initialized barrier.

`pthread_barrierattr_destroy()` Syntax

```
#include <pthread.h>

int  pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

`pthread_barrierattr_destroy()` Return Values

`pthread_barrierattr_destroy()` returns zero after completing successfully. Any other return value indicates that an error occurred. If the following condition occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Indicates that the value of *attr* was not valid.

Synchronization Across Process Boundaries

Each of the synchronization primitives can be used across process boundaries. The primitives are set up by ensuring that the synchronization variable is located in a shared memory segment and by calling the appropriate `init()` routine. The primitive must have been initialized with its shared attribute set to interprocess.

Producer and Consumer Problem Example

Example 4–17 shows the producer and consumer problem with the producer and consumer in separate processes. The main routine maps zero-filled memory shared with its child process into its address space.

A child process is created to run the consumer. The parent runs the producer.

This example also shows the drivers for the producer and consumer. The `producer_driver()` reads characters from `stdin` and calls `producer()`. The `consumer_driver()` gets characters by calling `consumer()` and writes them to `stdout`.

The data structure for Example 4–17 is the same as the structure used for the condition variables example, shown in Example 4–4. Two semaphores represent the number of full and empty buffers. The semaphores ensure that producers wait for empty buffers and that consumers wait until the buffers are full.

EXAMPLE 4–17 Synchronization Across Process Boundaries

```
main() {
    int zfd;
    buffer_t *buffer;
    pthread_mutexattr_t mattr;
    pthread_condattr_t cvattr_less, cvattr_more;

    zfd = open("/dev/zero", O_RDWR);
    buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
        PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);
    buffer->occupied = buffer->nextin = buffer->nextout = 0;

    pthread_mutex_attr_init(&mattr);
    pthread_mutexattr_setpshared(&mattr,
        PTHREAD_PROCESS_SHARED);

    pthread_mutex_init(&buffer->lock, &mattr);
    pthread_condattr_init(&cvattr_less);
    pthread_condattr_setpshared(&cvattr_less, PTHREAD_PROCESS_SHARED);
    pthread_cond_init(&buffer->less, &cvattr_less);
    pthread_condattr_init(&cvattr_more);
    pthread_condattr_setpshared(&cvattr_more,
        PTHREAD_PROCESS_SHARED);
    pthread_cond_init(&buffer->more, &cvattr_more);
```

EXAMPLE 4–17 Synchronization Across Process Boundaries *(Continued)*

```
    if (fork() == 0)
        consumer_driver(buffer);
    else
        producer_driver(buffer);
}

void producer_driver(buffer_t *b) {
    int item;

    while (1) {
        item = getchar();
        if (item == EOF) {
            producer(b, ‘\0’);
            break;
        } else
            producer(b, (char)item);
    }
}

void consumer_driver(buffer_t *b) {
    char item;

    while (1) {
        if ((item = consumer(b)) == ’\0’)
            break;
        putchar(item);
    }
}
```

Comparing Primitives

The most basic synchronization primitive in threads is the mutual exclusion lock. So, mutual exclusion lock is the most efficient mechanism in both memory use and execution time. The basic use of a mutual exclusion lock is to serialize access to a resource.

The next most efficient primitive in threads is the condition variable. The basic use of a condition variable is to block on a change of state. The condition variable provides a thread wait facility. Remember that a mutex lock must be acquired before blocking on a condition variable and must be unlocked after returning from `pthread_cond_wait()`. The mutex lock must also be held across the change of state that occurs before the corresponding call to `pthread_cond_signal()`.

The semaphore uses more memory than the condition variable. The semaphore is easier to use in some circumstances because a semaphore variable operates on state rather than on control. Unlike a lock, a semaphore does not have an owner. Any thread can increment a semaphore that has blocked.

The read-write lock permits concurrent reads and exclusive writes to a protected resource. The read-write lock is a single entity that can be locked in *read* or *write* mode. To modify a resource, a thread must first acquire the exclusive write lock. An exclusive write lock is not permitted until all read locks have been released.

◆ ◆ ◆ CHAPTER 5

Programming With the Oracle Solaris Software

This chapter describes how multithreading interacts with the Oracle Solaris software and how the software has changed to support multithreading.

- "Forking Issues in Process Creation" on page 145
- "Process Creation: exec and exit Issues" on page 149
- "Timers, Alarms, and Profiling" on page 149
- "Nonlocal Goto: setjmp and longjmp" on page 150
- "Resource Limits" on page 151
- "LWPs and Scheduling Classes" on page 151
- "Extending Traditional Signals" on page 153
- "I/O Issues" on page 162

Forking Issues in Process Creation

The default handling of `fork()` in the Solaris 9 product and earlier Oracle Solaris releases is somewhat different from the way `fork()` is handled in POSIX threads. For Oracle Solaris releases after Solaris 9, `fork()` behaves as specified for POSIX threads in all cases.

Table 5–1 compares the differences and similarities of `fork()` handling in Oracle Solaris threads and pthreads. When the comparable interface is not available either in POSIX threads or in Oracle Solaris threads, the '—' character appears in the table column.

TABLE 5–1 Comparing POSIX and Oracle Solaris `fork()` Handling

	Oracle Solaris Interface	POSIX Threads Interface

Fork-one model	`fork1(2)`
	`fork(2)`
		`fork(2)`

Fork-all model	`forkall(2)`	`forkall(2)`
Fork safety	—	`pthread_atfork(3C)`

Fork-One Model

As shown in Table 5–1, the behavior of the pthreads `fork(2)` function is the same as the behavior of the Oracle Solaris `fork1(2)` function. Both the pthreads `fork(2)` function and the Oracle Solaris `fork1(2)` function create a new process, duplicating the complete address space in the child. However, both functions duplicate only the calling thread in the child process.

Duplication of the calling thread in the child process is useful when the child process immediately calls `exec()`, which is what happens after most calls to `fork()`. In this case, the child process does not need a duplicate of any thread other than the thread that called `fork()`.

In the child, do not call any library functions after calling `fork()` and before calling `exec()`. One of the library functions might use a lock that was held in the parent at the time of the `fork()`. The child process may execute only Async-Signal-Safe operations until one of the `exec()` handlers is called. See "Signal Handlers and Async-Signal Safety" on page 159 for more information about Async-Signal-Safe functions.

Fork-One Safety Problem and Solution

Besides the usual concerns such as locking shared data, a library should be well behaved with respect to forking a child process when only the thread that called `fork()` is running. The problem is that the sole thread in the child process might try to grab a lock held by a thread not duplicated in the child.

Most programs are not likely to encounter this problem. Most programs call `exec()` in the child right after the return from `fork()`. However, if the program has to carry out actions in the child before calling `exec()`, or never calls `exec()`, then the child *could* encounter deadlocks. Each library writer should provide a safe solution, although not providing a fork-safe library is not a large concern because this condition is rare.

For example, assume that T1 is in the middle of printing something and holds a lock for `printf()`, when T2 forks a new process. In the child process, if the sole thread (T2) calls `printf()`, T2 promptly deadlocks.

The POSIX `fork()` or the Oracle Solaris `fork1()` function duplicates only the thread that calls `fork()` or `fork1()` . If you call Oracle Solaris `forkall()` to duplicate all threads, this issue is not a concern.

However, `forkall()` can cause other problems and should be used with care. For instance, if a thread calls `forkall()`, the parent thread performing I/O to a file is replicated in the child process. Both copies of the thread will continue performing I/O to the same file, one in the parent and one in the child, leading to malfunctions or file corruption.

To prevent deadlock when calling `fork1()`, ensure that no locks are being held at the time of forking. The most obvious way to prevent deadlock is to have the forking thread acquire all the locks that could possibly be used by the child. Because you cannot acquire all locks for `printf()` because `printf()` is owned by `libc`, you must ensure that `printf()` is not being used at `fork()` time.

Tip – The Thread Analyzer utility included in the Oracle Solaris Studio software enables you to detect deadlocks in a running program. See *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* for more information.

To manage the locks in your library, you should perform the following actions:

- Identify all the locks used by the library.
- Identify the locking order for the locks used by the library. If a strict locking order is not used, then lock acquisition must be managed carefully.
- Arrange to acquire all locks at fork time.

In the following example, the list of locks used by the library is { `L1,...Ln`}. The locking order for these locks is also `L1...Ln`.

```
mutex_lock(L1);
mutex_lock(L2);
fork1(...);
mutex_unlock(L1);
mutex_unlock(L2);
```

When using either Oracle Solaris threads or POSIX threads, you can add a call to `pthread_atfork(f1, f2, f3)` in your library's `.init()` section. The `f1()`, `f2()`, `f3()` are defined as follows:

```
f1() /* This is executed just before the process forks. */
{
 mutex_lock(L1); |
 mutex_lock(...); | -- ordered in lock order
 mutex_lock(Ln); |
 } V

f2() /* This is executed in the child after the process forks. */
 {
 mutex_unlock(L1);
 mutex_unlock(...);
 mutex_unlock(Ln);
 }

f3() /* This is executed in the parent after the process forks. */
 {
 mutex_unlock(L1);
 mutex_unlock(...);
 mutex_unlock(Ln);
 }
```

Virtual Forks–vfork

The standard `vfork(2)` function is unsafe in multithreaded programs. `vfork(2)` , like `fork1(2)`, copies only the calling thread in the child process. As in nonthreaded implementations, `vfork()` does not copy the address space for the child process.

Be careful that the thread in the child process does not change memory before the thread calls exec(2). vfork() gives the parent address space to the child. The parent gets its address space back after the child calls exec() or exits. The child must not change the state of the parent.

For example, disastrous problems occur if you create new threads between the call to vfork() and the call to exec().

Solution: pthread_atfork

Use pthread_atfork() to prevent deadlocks whenever you use the fork-one model.

```
#include <pthread.h>

int pthread_atfork(void (*prepare) (void), void (*
parent) (void),
    void (*child) (void) );
```

The pthread_atfork() function declares fork() handlers that are called before and after fork() in the context of the thread that called fork().

- The *prepare* handler is called before fork() starts.
- The *parent* handler is called after fork() returns in the parent.
- The *child* handler is called after fork() returns in the child.

Any handler argument can be set to NULL. The order in which successive calls to pthread_atfork() are made is significant.

For example, a *prepare* handler could acquire all the mutexes needed. Then the *parent* and *child* handlers could release the mutexes. The *prepare* handler acquiring all required mutexes ensures that all relevant locks are held by the thread calling the fork function *before* the process is forked. This technique prevents a deadlock in the child.

See the pthread_atfork(3C) man page for more information.

Fork-All Model

The Oracle Solaris forkall(2) function duplicates the address space and all the threads in the child. Address space duplication is useful, for example, when the child process never calls exec(2) but does use its copy of the parent address space.

When one thread in a process calls Oracle Solaris forkall(2), threads that are blocked in an interruptible system call will return EINTR.

Be careful not to create locks that are held by both the parent and child processes. Locks held in both parent and child processes occur when locks are allocated in shared memory by calling mmap() with the MAP_SHARED flag. This problem does not occur if the fork-one model is used.

Choosing the Right Fork

Starting with the Oracle Solaris 10 release, a call to fork() is identical to a call to fork1(). Specifically, only the calling thread is replicated in the child process. The behavior is the same as the POSIX fork().

In previous releases of the Oracle Solaris software, the behavior of fork() was dependent on whether the application was linked with the POSIX threads library. When linked with -lthread (Oracle Solaris threads) but not linked with -lpthread (POSIX threads), fork() was the same as forkall(). When linked with -lpthread, regardless of whether fork() was also linked with -lthread , fork() was the same as fork1().

Starting with the Oracle Solaris 10 release, neither -lthread nor -lpthread is required for multithreaded applications. The -mt option is used to indicate that you are compiling a multithreaded application. The standard C library provides all threading support for both sets of application program interfaces. Applications that require *replicate all* fork semantics must call forkall().

Process Creation: exec and exit Issues

Both the exec(2) and exit(2) system calls work as these functions do in single-threaded processes with the following exception. In a multithreaded application, the functions destroy all the threads in the address space. Both calls block until all the execution resources, and so all active threads, are destroyed.

When exec() rebuilds the process, exec() creates a single lightweight process (LWP). The process startup code builds the initial thread. As usual, if the initial thread returns, the thread calls exit() and the process is destroyed.

When all the threads in a process exit, the process exits. A call to any exec() function from a process with more than one thread terminates all threads, and loads and executes the new executable image. No destructor functions are called.

Timers, Alarms, and Profiling

Over several releases, the Oracle Solaris OS has evolved to a per-process mode for alarms, interval timers, and profiling.

Timers

All timers are per-process except for the real time profile interval timer, which is per_LWP. See the setitimer(2) man page for a description of the ITIMER_REALPROF timer.

The timer IDs of per-process timers are usable from any LWP. The expiration signals are generated for the process rather than directed to a specific LWP.

The per-process timers are deleted only by `timer_delete`(3RT), or when the process terminates.

In the Oracle Solaris 11.1 release, the POSIX timer API has been enhanced to provide a thread (LWP) directed signal. The thread directed signal notification can be requested with the use of the `SIGEV_SIGNAL_THR` sigevent type. You can effectively create per thread timers by using this sigevent type. See the `timer_create`(3C) man page for more information.

Alarms

Alarms operate at the process level, not at the thread level. The `alarm()` function sends the signal `SIGALRM` to the calling process rather than the calling thread.

Profiling a Multithreaded Program

The `profil()` system call for multithreaded processes has global impact on all LWPs and threads in the process. Threads cannot use `profil()` for individual thread profiling. See the `profil`(2) man page for more information.

Tip – The Performance Analyzer tool, included in the Oracle Solaris Studio software, can be used for extensive profiling of multithreaded and single threaded programs. The tool enables you to see in detail what a thread is doing at any given point. See the *Oracle Solaris Studio 12.3: Performance Analyzer* for more information.

Nonlocal Goto: setjmp and longjmp

The scope of `setjmp()` and `longjmp()` is limited to one thread, which is acceptable most of the time. However, the limited scope does mean that a thread that handles a signal can execute a `longjmp()` only when a `setjmp()` is performed in the same thread.

Resource Limits

Resource limits are set on the entire process and are determined by adding the resource use of all threads in the process. When a soft resource limit is exceeded, the offending thread is sent the appropriate signal. The sum of the resources that are used in the process is available through `getrusage`(3C).

LWPs and Scheduling Classes

The Oracle Solaris kernel has three ranges of dispatching priority. The highest-priority range (100 to 159) corresponds to the Realtime (RT) scheduling class. The middle-priority range (60 to 99) corresponds to the system (SYS) scheduling class. The system class cannot be applied to a user process. The lowest-priority range (0 to 59) is shared by the timesharing (TS), interactive (IA), fair-share (FSS), and fixed priority (FX) scheduling classes.

A scheduling class is maintained for each LWP. When a process is created, the initial LWP inherits the scheduling class and priority of the creating LWP in the parent process. As more threads are created, their associated LWPs also inherit this scheduling class and priority.

Threads have the scheduling class and priority of their underlying LWPs. Each LWP in a process can have a unique scheduling class and priority that are visible to the kernel.

Thread priorities regulate contention for synchronization objects. By default, LWPs are in the timesharing class. For compute-bound multithreading, thread priorities are not very useful. For multithreaded applications that use the MT libraries to do synchronization frequently, thread priorities are more meaningful.

The scheduling class is set by `priocntl`(2). How you specify the first two arguments determines whether only the calling LWP or all the LWPs of one or more processes are affected. The third argument of `priocntl()` is the command, which can be one of the following commands.

- `PC_GETCID` - Get the class ID and class attributes for a specific class.
- `PC_GETCLINFO` - Get the class name and class attributes for a specific class.
- `PC_GETPARMS` - Get the class identifier and the class-specific scheduling parameters of a process, an LWP with a process, or a group of processes.
- `PC_SETPARMS` - Set the class identifier and the class-specific scheduling parameters of a process, an LWP with a process, or a group of processes.

The user-level priority of an LWP is its priority within its class, not its dispatch priority. This does not change over time except by the application of the `priocntl()` system call. The kernel determines the dispatch priority of an LWP based on its scheduling class, its priority within that class, and possibly other factors such as its recently-used CPU time.

Timeshare Scheduling

Timeshare scheduling attempts to distribute processor resources fairly among the LWPs in the timesharing (TS) and interactive (IA) scheduling classes.

The `priocntl(2)` call sets the class priority of one or more processes or LWPs. The normal range of timesharing class priorities is -60 to +60. The higher the value, the higher the kernel dispatch priority. The default timesharing class priority is 0.

The old concept of a `nice` value for a process, where a lower `nice` value means a higher priority, is maintained for all of the TS, IA, and FSS scheduling classes. The old `nice`-based `setpriority(3C)` and `nice(2)` interfaces continue to work by mapping `nice` values into priority values. Setting a `nice` value changes the priority and vice-versa. The range of `nice` values is -20 to +20. A `nice` value of 0 corresponds to a priority of 0. A `nice` value of -20 corresponds to a priority of +60.

The dispatch priority of time-shared LWPs is calculated from the instantaneous CPU use rate of the LWP and from its class priority. The class priority indicates the relative priority of the LWPs to the timeshare scheduler.

LWPs with a smaller class priority value get a smaller, but nonzero, share of the total processing. An LWP that has received a larger amount of processing is given lower dispatch priority than an LWP that has received little or no processing.

Realtime Scheduling

The Realtime class (`RT`) can be applied to a whole process or to one or more LWPs in a process. You must have superuser privilege to use the Realtime class.

The normal range of realtime class priorities is 0 to 59. The dispatch priority of an LWP in the realtime class is fixed at its class priority plus 100.

The scheduler always dispatches the highest-priority Realtime LWP. The high-priority Realtime LWP preempts a lower-priority LWP when a higher-priority LWP becomes runnable. A preempted LWP is placed at the head of its level queue.

A Realtime LWP retains control of a processor until the LWP is preempted, the LWP suspends, or its Realtime priority is changed. LWPs in the RT class have absolute priority over processes in the TS class.

A new LWP inherits the scheduling class of the parent process or LWP. An RT class LWP inherits the parent's time slice, whether finite or infinite.

A finite time slice LWP runs until the LWP terminates, blocks on an I/O event, gets preempted by a higher-priority runnable Realtime process, or the time slice expires.

An LWP with an infinite time slice ceases execution only when the LWP terminates, blocks, or is preempted.

Fair Share Scheduling

The fair share scheduler (FSS) scheduling class allows allocation of CPU time based on shares.

The normal range of fair share scheduler class priorities is -60 to 60, which get mapped by the scheduler into dispatch priorities in the same range (0 to 59) as the TS and IA scheduling classes. All LWPs in a process must run in the same scheduling class. The FSS class schedules individual LWPs, not whole processes. Thus, a mix of processes in the FSS and TS/IA classes could result in unexpected scheduling behavior in both cases.

The TS/IA or the FSS scheduling class processes do not compete for the same CPUs. Processor sets enable mixing TS/IA with FSS in a system. However, all processes in each processor set must be in either the TS/IA or the FSS scheduling class.

Fixed Priority Scheduling

The FX, fixed priority, scheduling class assigns fixed priorities and time quantum not adjusted to accommodate resource consumption. Process priority can be changed only by the process that assigned the priority or an appropriately privileged process. For more information about FX, see the `priocntl`(1) and `dispadmin`(1M) man pages.

The normal range of fixed priority scheduler class priorities is 0 to 60, which get mapped by the scheduler into dispatch priorities in the same range (0 to 59) as the TS and IA scheduling classes.

Extending Traditional Signals

The traditional UNIX signal model is extended to threads in a fairly natural way. The key characteristics are that the signal disposition is process-wide, but the signal mask is per-thread. The process-wide disposition of signals is established using the traditional mechanisms `signal`(3C),`sigaction`(2) , and so on.

When a signal handler is marked `SIG_DFL` or `SIG_IGN`, the action on receipt of a signal is performed on the entire receiving process. These signals include exit, core dump, stop, continue, and ignore. The action on receipt of these signals is carried out on all threads in the process. Therefore, the issue of which thread picks the signal is nonexistent. The exit, core dump, stop, continue, and ignore signals have no handlers. See the `signal.h`(3HEAD) man page for basic information about signals.

Each thread has its own signal mask. The signal mask lets a thread block some signals while the thread uses memory or another state that is also used by a signal handler. All threads in a process share the set of signal handlers that are set up by `sigaction`(2) and its variants.

A thread in one process cannot send a signal to a specific thread in another process. A signal sent by kill(2), sigsend(2), or sigqueue(3RT) to a process is handled by any receptive threads in the process.

Signals are divided into the following categories: traps, exceptions, and interrupts. Traps and exceptions are synchronously generated signals. Interrupts are asynchronously generated signals.

As in traditional UNIX, if a signal is pending, additional occurrences of that signal normally have no additional effect. A pending signal is represented by a bit, not by a counter. However, signals that are posted through the sigqueue(3RT) interface allow multiple instances of the same signal to be queued to the process.

As is the case with single-threaded processes, when a thread receives a signal while blocked in a system call, the thread might return early. When a thread returns early, the thread either returns an EINTR error code, or, in the case of I/O calls, with fewer bytes transferred than requested.

Of particular importance to multithreaded programs is the effect of signals on pthread_cond_wait(3C). This call usually returns without error, a return value of zero, only in response to a pthread_cond_signal(3C) or a pthread_cond_broadcast(3C). However, if the waiting thread receives a traditional UNIX signal, pthread_cond_wait() returns with a return value of zero even though the wakeup was spurious.

Synchronous Signals

Traps, such as SIGILL, SIGFPE, and SIGSEGV, result from an operation on the thread, such as dividing by zero or making reference to nonexistent memory. A trap is handled only by the thread that caused the trap. Several threads in a process can generate and handle the same type of trap simultaneously.

The idea of signals to individual threads is easily extended for synchronously generated signals. The handler is invoked on the thread that generated the synchronous signal.

However, if the process chooses not to establish an appropriate signal handler, the default action is taken when a trap occurs. The default action occurs even if the offending thread is blocked on the generated signal. The default action for such signals is to terminate the process, perhaps with a core dump.

Such a synchronous signal usually means that something is seriously wrong with the whole process, not just with a thread. In this case, terminating the process is often a good choice.

Asynchronous Signals

Interrupts, such as SIGINT and SIGIO, are asynchronous with any thread and result from some action outside the process. These interrupts might be signals sent explicitly by another process, or might represent external actions such as a user typing a Control-C.

An interrupt can be handled by any thread whose signal mask allows the interrupt. When more than one thread is able to receive the interrupt, only one thread is chosen.

When multiple occurrences of the same signal are sent to a process, then each occurrence can be handled by a separate thread. However, the available threads must not have the signal masked. When all threads have the signal masked, then the signal is marked *pending* and the first thread to unmask the signal handles the signal.

Continuation Semantics

Continuation semantics are the traditional way to deal with signals. When a signal handler returns, control resumes where the process was at the time of the interruption. This control resumption is well suited for asynchronous signals in single-threaded processes, as shown in Example 5–1.

This control resumption is also used as the exception-handling mechanism in other programming languages, such as PL/1.

EXAMPLE 5–1 Continuation Semantics

```
unsigned int nestcount;

unsigned int A(int i, int j) {
    nestcount++;

    if (i==0)
        return(j+1)
    else if (j==0)
        return(A(i-1, 1));
    else
        return(A(i-1, A(i, j-1)));
}

void sig(int i) {
    printf("nestcount = %d\n", nestcount);
}

main() {
    sigset(SIGINT, sig);
    A(4,4);
}
```

Operations on Signals

This section describes the operations on signals.

“Setting the Thread's Signal Mask” on page 156
“Sending a Signal to a Specific Thread” on page 156

"Waiting for a Specified Signal" on page 156
"Waiting for Specified Signal Within a Given Time" on page 157

Setting the Thread's Signal Mask

`pthread_sigmask(3C)` does for a thread what `sigprocmask(2)` does for a process. `pthread_sigmask()` sets the thread's signal mask. When a new thread is created, its initial mask is inherited from its creator.

The call to `sigprocmask()` in a multithreaded process is equivalent to a call to `pthread_sigmask()`. See the `sigprocmask(2)` man page for more information.

Sending a Signal to a Specific Thread

`pthread_kill(3C)` is the thread analog of `kill(2)`. A `pthread_kill()` call sends a signal to a specific thread. A signal that is sent to a specified thread is different from a signal that is sent to a process. When a signal is sent to a process, the signal can be handled by any thread in the process. A signal sent by `pthread_kill()` can be handled only by the specified thread.

You can use `pthread_kill()` to send signals only to threads in the current process. Because the thread identifier, type *thread_t*, is local in scope, you cannot name a thread outside the scope of the current process.

On receipt of a signal by the target thread, the action invoked (handler, `SIG_DFL`, or `SIG_IGN`) is global, as usual. If you send `SIG`*XXX* to a thread, and `SIG`*XXX* to kill a process, the whole process is killed when the target thread receives the signal.

Waiting for a Specified Signal

For multithreaded programs, `sigwait(2)` is the preferred interface to use because `sigwait()` deals well with asynchronously generated signals.

`sigwait()` causes the calling thread to wait until any signal identified by the `sigwait()` function's *set* argument is delivered to the thread. While the thread is waiting, signals identified by the *set* argument are unmasked, but the original mask is restored when the call returns.

All signals identified by the *set* argument must be blocked on all threads, including the calling thread. Otherwise, `sigwait()` might not work correctly.

Use `sigwait()` to separate threads from asynchronous signals. You can create one thread that listens for asynchronous signals while you create other threads to block any asynchronous signals set to this process.

The following example shows the syntax of `sigwait()`.

```
#include <signal.h>
int sigwait(const sigset_t *set, int *sig
);
```

When the signal is delivered, `sigwait()` clears the pending signal and places the signal number in *sig*. Many threads can call `sigwait()` at the same time, but only one thread returns for each signal that is received.

With `sigwait()`, you can treat asynchronous signals synchronously. A thread that deals with such signals calls `sigwait()` and returns as soon as a signal arrives. By ensuring that all threads, including the caller of `sigwait()`, mask asynchronous signals, ensures signals are handled only by the intended handler and are handled safely.

By always masking all signals in all threads and calling `sigwait()` as necessary, your application is much safer for threads that depend on signals.

Usually, you create one or more threads that call `sigwait()` to wait for signals. Because `sigwait()` retrieves even masked signals, be sure to block the signals of interest in all other threads so the signals are not accidentally delivered.

When a signal arrives, a signal-handling thread returns from `sigwait()` , handles the signal, and calls `sigwait()` again to wait for more signals. The signal-handling thread is not restricted to using Async-Signal-Safe functions. The signal-handling thread can synchronize with other threads in the usual way. The Async-Signal-Safe category is defined in "MT Interface Safety Levels" on page 206.

Note – `sigwait()` cannot receive synchronously generated signals.

Waiting for Specified Signal Within a Given Time

`sigtimedwait`(3RT) is similar to `sigwait`(2) except that `sigtimedwait()` fails and returns an error when a signal is not received in the indicated amount of time. See the `sigtimedwait`(3RT) man page for more information.

Thread-Directed Signals

The UNIX signal mechanism is extended with the idea of thread-directed signals. Thread-directed signals are just like ordinary asynchronous signals, except that thread-directed signals are sent to a particular thread instead of to a process.

A separate thread that waits for asynchronous signals can be safer and easier than installing a signal handler that processes the signals.

A better way to deal with asynchronous signals is to treat these signals synchronously. By calling `sigwait(2)`, a thread can wait until a signal occurs. See "Waiting for a Specified Signal" on page 156.

EXAMPLE 5–2 Asynchronous Signals and sigwait(2)

```
main() {
    sigset_t set;
    void runA(void);
    int sig;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    pthread_sigmask(SIG_BLOCK, &set, NULL);
    pthread_create(NULL, 0, runA, NULL, PTHREAD_DETACHED, NULL);

    while (1) {
        sigwait(&set, &sig);
        printf("nestcount = %d\n", nestcount);
        printf("received signal %d\n", sig);
    }
}

void runA() {
    A(4,4);
    exit(0);
}
```

This example modifies the code of Example 5–1. The main routine masks the `SIGINT` signal, creates a child thread that calls function A of the previous example, and issues `sigwait()` to handle the `SIGINT` signal.

Note that the signal is masked in the compute thread because the compute thread inherits its signal mask from the main thread. The main thread is protected from `SIGINT` while, and only while, the thread is not blocked inside of `sigwait()`.

Also, note that no danger exists of having system calls interrupted when you use `sigwait()`.

Completion Semantics

Another way to deal with signals is with completion semantics.

Use completion semantics when a signal indicates that something so catastrophic has happened that no reason exists to continue executing the current code block. The signal handler runs instead of the remainder of the block that had the problem. In other words, the signal handler completes the block.

In Example 5–3, the block in question is the body of the `then` part of the `if` statement. The call to `setjmp(3C)` saves the current register state of the program in *jbuf* and returns 0, thereby executing the block.

EXAMPLE 5–3 Completion Semantics

```
sigjmp_buf jbuf;
void mult_divide(void) {
    int a, b, c, d;
```

EXAMPLE 5–3 Completion Semantics *(Continued)*

```
    void problem();

    sigset(SIGFPE, problem);
    while (1) {
        if (sigsetjmp(&jbuf) == 0) {
            printf("Three numbers, please:\n");
            scanf("%d %d %d", &a, &b, &c);
            d = a*b/c;
            printf("%d*%d/%d = %d\n", a, b, c, d);
        }
    }
}

void problem(int sig) {
    printf("Couldn't deal with them, try again\n");
    siglongjmp(&jbuf, 1);
}
```

If a `SIGFPE` floating-point exception occurs, the signal handler is invoked.

The signal handler calls `siglongjmp(3C)`, which restores the register state saved in *jbuf*, causing the program to return from `sigsetjmp()` again. The registers that are saved include the program counter and the stack pointer.

This time, however, `sigsetjmp(3C)` returns the second argument of `siglongjmp()`, which is 1. Notice that the block is skipped over, only to be executed during the next iteration of the `while` loop.

You can use `sigsetjmp(3C)` and `siglongjmp(3C)` in multithreaded programs. Be careful that a thread never does a `siglongjmp()` that uses the results of another thread's `sigsetjmp()`.

Also, `sigsetjmp()` and `siglongjmp()` restore as well as save the signal mask, but `setjmp(3C)` and `longjmp(3C)` do not.

Use `sigsetjmp()` and `siglongjmp()` when you work with signal handlers.

Completion semantics are often used to deal with exceptions. In particular, the Oracle Ada programming language uses this model.

Note – Remember, `sigwait`(2) should *never* be used with synchronous signals.

Signal Handlers and Async-Signal Safety

A concept that is similar to thread safety is Async-Signal safety. Async-Signal-Safe operations are guaranteed not to interfere with operations that are being interrupted.

The problem of Async-Signal safety arises when the actions of a signal handler can interfere with the operation that is being interrupted.

For example, suppose a program is in the middle of a call to `printf`(3C), and a signal occurs whose handler calls `printf()`. In this case, the output of the two `printf()` statements would be intertwined. To avoid the intertwined output, the handler should not directly call `printf()` when `printf()` might be interrupted by a signal.

This problem cannot be solved by using synchronization primitives. Any attempt to synchronize between the signal handler and the operation being synchronized would produce an immediate deadlock.

Suppose that `printf()` is to protect itself by using a mutex. Now, suppose that a thread that is in a call to `printf()` and so holds the lock on the mutex is interrupted by a signal.

If the handler calls `printf()`, the thread that holds the lock on the mutex attempts to take the mutex again. Attempting to take the mutex results in an instant deadlock.

To avoid interference between the handler and the operation, ensure that the situation never arises. Perhaps you can mask off signals at critical moments, or invoke only Async-Signal-Safe operations from inside signal handlers.

The only routines that POSIX guarantees to be Async-Signal-Safe are listed in Table 5–2. Any signal handler can safely call in to one of these functions.

TABLE 5–2 Async-Signal-Safe Functions

`_Exit()`	`fpathconf()`	`read()`	`sigset()`
`_exit()`	`fstat()`	`readlink()`	`sigsuspend()`
`abort()`	`fsync()`	`recv()`	`sockatmark()`
`accept()`	`ftruncate()`	`recvfrom()`	`socket()`
`access()`	`getegid()`	`recvmsg()`	`socketpair()`
`aio_error()`	`geteuid()`	`rename()`	`stat()`
`aio_return()`	`getgid()`	`rmdir()`	`symlink()`
`aio_suspend()`	`getgroups()`	`select()`	`sysconf()`
`alarm()`	`getpeername()`	`sem_post()`	`tcdrain()`
`bind()`	`getpgrp()`	`send()`	`tcflow()`
`cfgetispeed()`	`getpid()`	`sendmsg()`	`tcflush()`
`cfgetospeed()`	`getppid()`	`sendto()`	`tcgetattr()`
`cfsetispeed()`	`getsockname()`	`setgid()`	`tcgetattr()`
`cfsetospeed()`	`getsockopt()`	`setpgid()`	`tcsendbreak()`
`chdir()`	`getuid()`	`setsid()`	`tcsetattr()`

TABLE 5–2 Async-Signal-Safe Functions (Continued)

chmod()	kill()	setsockopt()	tcsetpgrp()
chown()	link()	setuid()	time()
clock_gettime()	listen()	shutdown()	timer_getoverrun()
close()	lseek()	sigaction()	timer_gettime()
connect()	lstat()	sigaddset()	timer_settime()
creat()	mkdir()	sigdelset()	times()
dup()	mkfifo()	sigemptyset()	umask()
dup2()	open()	sigfillset()	uname()
execle()	pathconf()	sigismember()	ulink()
execve()	pause()	sleep()	utime()
fchmod()	pipe()	signal()	wait()
fchown()	poll()	sigpause()	waitpid()
fcntl()	posix_trace_event()	sigpending()	write()
fdatasync()	pselect()	sigprocmask()	
fork()	raise()	sigqueue()	

Interrupted Waits on Condition Variables

When an unmasked caught signal is delivered to a thread waiting on a condition variable, when the signal handler returns, the thread returns from the condition wait function with a spurious wakeup: `pthread_cond_wait()` and `pthread_cond_timedwait()` return 0 even though no call to `pthread_cond_signal()` or `pthread_cond_broadcast()` was made by another thread. Whether `SA_RESTART` has been specified as a flag to `sigaction()` has no effect here. The `pthread_cond_wait()` and `pthread_cond_timedwait()` functions are not automatically restarted. In all cases, the associated mutex lock is reacquired before returning from the condition wait.

Re-acquisition of the associated mutex lock does not imply that the mutex is locked while the thread is executing the signal handler. The state of the mutex in the signal handler is undefined.

I/O Issues

One of the attractions of multithreaded programming is I/O performance. The traditional UNIX API gave you little assistance in this area. You either used the facilities of the file system or bypassed the file system entirely.

This section shows how to use threads to get more flexibility through I/O concurrency and multibuffering. This section also discusses the differences and similarities between the approaches of synchronous I/O with threads, and asynchronous I/O with and without threads.

I/O as a Remote Procedure Call

In the traditional UNIX model, I/O appears to be synchronous, as if you were placing a remote procedure call to the I/O device. Once the call returns, then the I/O has completed, or at least appears to have completed. A write request, for example, might merely result in the transfer of the data to a buffer in the operating environment.

The advantage of this model is familiar concept of procedure calls.

An alternative approach not found in traditional UNIX systems is the asynchronous model, in which an I/O request merely starts an operation. The program must somehow discover when the operation completes.

The asynchronous model is not as simple as the synchronous model. But, the asynchronous model has the advantage of allowing concurrent I/O and processing in traditional, single-threaded UNIX processes.

Tamed Asynchrony

You can get most of the benefits of asynchronous I/O by using synchronous I/O in a multithreaded program. With asynchronous I/O, you would issue a request and check later to determine when the I/O completes. You can instead have a separate thread perform the I/O synchronously. The main thread can then check for the completion of the operation at some later time perhaps by calling `pthread_join`(3C).

Asynchronous I/O

In most situations, asynchronous I/O is not required because its effects can be achieved with the use of threads, with each thread execution of synchronous I/O. However, in a few situations, threads cannot achieve what asynchronous I/O can.

The most straightforward example is writing to a tape drive to make the tape drive stream. Streaming prevents the tape drive from stopping while the drive is being written to. The tape moves forward at high speed while supplying a constant stream of data that is written to tape.

To support streaming, the tape driver in the kernel should use threads. The tape driver in the kernel must issue a queued write request when the tape driver responds to an interrupt. The interrupt indicates that the previous tape-write operation has completed.

Threads cannot guarantee that asynchronous writes are ordered because the order in which threads execute is indeterminate. You cannot, for example, specify the order of a write to a tape.

Asynchronous I/O Operations

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);

int aio_write(struct aiocb *aiocbp);

int aio_error(const struct aiocb *aiocbp);

ssize_t aio_return(struct aiocb *aiocbp);

int aio_suspend(struct aiocb *list[], int nent,
    const struct timespec *timeout);

int aio_waitn(struct aiocb *list[], uint_t nent, uint_t *nwait,
    const struct timespec *timeout);

int aio_cancel(int fildes, struct aiocb *aiocbp);
```

`aio_read`(3RT) and `aio_write`(3RT) are similar in concept to `pread`(2) and `pwrite`(2), except that the parameters of the I/O operation are stored in an asynchronous I/O control block (`aiocbp`) that is passed to `aio_read()` or `aio_write()`:

```
    aiocbp->aio_fildes;    /* file descriptor */
    aiocbp->aio_buf;       /* buffer */
    aiocbp->aio_nbytes;    /* I/O request size */
    aiocbp->aio_offset;    /* file offset */
```

In addition, if desired, an asynchronous notification type (most commonly a queued signal) can be specified in the '`struct sigevent`' member:

```
    aiocbp->aio_sigevent;  /* notification type */
```

A call to `aio_read()` or `aio_write()` results in the initiation or queueing of an I/O operation. The call returns without blocking.

The `aiocbp` value may be used as an argument to `aio_error`(3RT) and `aio_return`(3RT) in order to determine the error status and return status of the asynchronous operation while it is proceeding.

Waiting for I/O Operation to Complete

You can wait for one or more outstanding asynchronous I/O operations to complete by calling `aio_suspend()` or `aio_waitn()`. Use `aio_error()` and `aio_return()` on the completed asynchronous I/O control blocks to determine the success or failure of the I/O operation.

The aio_suspend() and aio_waitn() functions take a *timeout* argument, which indicates how long the caller is willing to wait. A NULL pointer means that the caller is willing to wait indefinitely. A pointer to a structure containing a zero value means that the caller is unwilling to wait at all.

You might start an asynchronous I/O operation, do some work, then call aio_suspend() or aio_waitn() to wait for the request to complete. Or you can rely on the asynchronous notification event specified in aio_sigevent() to occur to notify you when the operation completes.

Finally, a pending asynchronous I/O operation can be cancelled by calling aio_cancel(). This function is called with the address of the I/O control block that was used to initiate the I/O operation.

Shared I/O and New I/O System Calls

When multiple threads perform concurrent I/O operations with the same file descriptor, you might discover that the traditional UNIX I/O interface is not thread safe. The problem occurs with nonsequential I/O where the lseek(2) system call sets the file offset. The file offset is then used in the next read(2) or write(2) call to indicate where in the file the operation should start. When two or more threads are issuing an lseek() to the same file descriptor, a conflict results.

To avoid this conflict, use the pread() and pwrite() system calls.

```
#include <sys/types.h>
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);

ssize_t pwrite(int filedes, void *buf, size_t nbyte,
    off_t offset);
```

pread(2) and pwrite(2) behave just like read (2) and write(2) except that pread(2) and pwrite(2) take an additional argument, the file offset. With this argument, you specify the offset without using lseek(2), so multiple threads can use these routines safely for I/O on the same file descriptor.

Alternatives to getc and putc

An additional problem occurs with standard I/O. Programmers are accustomed to routines, such as getc(3C) and putc(3C) , that are implemented as macros, being very quick. Because of the speed of getc(3C) and putc(3C), these macros can be used within the inner loop of a program with no concerns about efficiency.

However, when getc(3C) and putc(3C) are made thread safe the macros suddenly become more expensive. The macros now require at least two internal subroutine calls, to lock and unlock a mutex.

To get around this problem, alternative versions of these routines are supplied: getc_unlocked(3C) and putc_unlocked(3C).

getc_unlocked(3C) and putc_unlocked(3C) do not acquire locks on a mutex. These getc_unlocked() or putc_unlocked() macros are as quick as the original, nonthread-safe versions of getc(3C) and putc(3C).

However, to use these macros in a thread-safe way, you must explicitly lock and release the mutexes that protect the standard I/O streams, using flockfile(3C) and funlockfile(3C). The calls to these latter routines are placed outside the loop. Calls to getc_unlocked() or putc_unlocked() are placed inside the loop.

New System Calls For Reliable Multithreaded Programming

In the Oracle Solaris 11 release, the following new APIs and flags have been added to make multithreaded programs more reliable:

*at() functions – Consists of versions of file handling system calls which take the working directory for relative pathnames as an argument, to avoid races with chdir(2) calls between multiple threads.

- openat(2)
- fchownat(2)
- fstatat(2)
- utimensat(2)
- unlinkat(2)
- faccessat(2)
- fchmodat(2)
- linkat(2)
- mkdirat(2)
- mknodat(2)
- readlinkat(2)
- renameat(2)
- symlinkat(2)

Flags – The following flags close a race condition between the open(2), dup(2), and dup2(2) calls and a following call to fcntl(2) function to set the FD_CLOEXEC flag by setting it atomically in the creation system call.

- O_CLOEXEC flag to open(2)
- F_DUPFD_CLOEXEC() and F_DUP2FD_CLOEXEC() arguments to fcntl(2)

CHAPTER 6

Programming With Oracle Solaris Threads

This chapter compares the application programming interface (API) for Oracle Solaris and POSIX threads, and explains the Oracle Solaris features that are not found in POSIX threads. The chapter discusses the following topics:

- "Comparing APIs for Oracle Solaris Threads and POSIX Threads" on page 167
- "Unique Oracle Solaris Threads Functions" on page 172
- "Similar Synchronization Functions: Read-Write Locks" on page 173
- "Similar Oracle Solaris Threads Functions" on page 179
- "Similar Synchronization Functions: Mutual Exclusion Locks" on page 190
- "Similar Synchronization Functions: Condition Variables" on page 194
- "Similar Synchronization Functions: Semaphores" on page 199
- "Special Issues for `fork()` and Oracle Solaris Threads" on page 204

Comparing APIs for Oracle Solaris Threads and POSIX Threads

The Oracle Solaris threads API and the pthreads API are two solutions to the same problem: build parallelism into application software. Although each API is complete, you can safely mix Oracle Solaris threads functions and pthread functions in the same program.

The two APIs do not match exactly, however. Oracle Solaris threads support functions that are not found in pthreads, and pthreads include functions that are not supported in the Oracle Solaris interface. For those functions that *do* match, the associated arguments might not, although the information content is effectively the same.

By combining the two APIs, you can use features not found in one API to enhance the other API. Similarly, you can run applications that use Oracle Solaris threads exclusively with applications that use pthreads exclusively on the same system.

Major API Differences

Oracle Solaris threads and pthreads are very similar in both API action and syntax. The major differences are listed in Table 6–1 .

TABLE 6–1 Unique Oracle Solaris Threads and pthreads Features

Oracle Solaris Threads	POSIX Threads
`thr_` prefix for threads function names, `sema_` prefix for semaphore function names	`pthread_` prefix for pthreads function names, `sem_` prefix for semaphore function names
Ability to create "daemon" threads	Cancellation semantics
Suspending and continuing a thread	Scheduling policies

Function Comparison Table

The following table compares Oracle Solaris threads functions with pthreads functions. Note that even when Oracle Solaris threads and pthreads functions appear to be essentially the same, the arguments to the functions can differ.

When a comparable interface is not available either in pthreads or Oracle Solaris threads, a hyphen '-' appears in the column. Entries in the pthreads column that are followed by (3RT) are functions in `librt`, the POSIX.1b Realtime Extensions library, which is not part of pthreads. Functions in this library provide most of the interfaces specified by the POSIX.1b Realtime Extension.

TABLE 6–2 Oracle Solaris Threads and POSIX pthreads Comparison

Oracle Solaris Threads	pthreads
`thr_create()`	`pthread_create()`
`thr_exit()`	`pthread_exit()`
`thr_join()`	`pthread_join()`
`thr_yield()`	`sched_yield()`(3RT)
`thr_self()`	`pthread_self()`
`thr_kill()`	`pthread_kill()`
`thr_sigsetmask()`	`pthread_sigmask()`
`thr_setprio()`	`pthread_setschedparam()`
`thr_getprio()`	`pthread_getschedparam()`
`thr_setconcurrency()`	`pthread_setconcurrency()`

TABLE 6–2 Oracle Solaris Threads and POSIX pthreads Comparison *(Continued)*

Oracle Solaris Threads	pthreads
`thr_getconcurrency()`	`pthread_getconcurrency()`
`thr_suspend()`	-
`thr_continue()`	-
`thr_keycreate()`	`pthread_key_create()`
-	`pthread_key_delete()`
`thr_setspecific()`	`pthread_setspecific()`
`thr_getspecific()`	`pthread_getspecific()`
-	`pthread_once()`
-	`pthread_equal()`
-	`pthread_cancel()`
-	`pthread_testcancel()`
-	`pthread_cleanup_push()`
-	`pthread_cleanup_pop()`
-	`pthread_setcanceltype()`
-	`pthread_setcancelstate()`
`mutex_lock()`	`pthread_mutex_lock()`
`mutex_unlock()`	`pthread_mutex_unlock()`
`mutex_trylock()`	`pthread_mutex_trylock()`
`mutex_init()`	`pthread_mutex_init()`
`mutex_destroy()`	`pthread_mutex_destroy()`
`cond_wait()`	`pthread_cond_wait()`
`cond_timedwait()`	`pthread_cond_timedwait()`
`cond_reltimedwait()`	`pthread_cond_reltimedwait_np()`
`cond_signal()`	`pthread_cond_signal()`
`cond_broadcast()`	`pthread_cond_broadcast()`
`cond_init()`	`pthread_cond_init()`
`cond_destroy()`	`pthread_cond_destroy()`
`rwlock_init()`	`pthread_rwlock_init()`

TABLE 6–2 Oracle Solaris Threads and POSIX pthreads Comparison *(Continued)*

Oracle Solaris Threads	pthreads
`rwlock_destroy()`	`pthread_rwlock_destroy()`
`rw_rdlock()`	`pthread_rwlock_rdlock()`
`rw_wrlock()`	`pthread_rwlock_wrlock()`
`rw_unlock()`	`pthread_rwlock_unlock()`
`rw_tryrdlock()`	`pthread_rwlock_tryrdlock()`
`rw_trywrlock()`	`pthread_rwlock_trywrlock()`
-	`pthread_rwlockattr_init()`
-	`pthread_rwlockattr_destroy()`
-	`pthread_rwlockattr_getpshared()`
-	`pthread_rwlockattr_setpshared()`
`sema_init()`	`sem_init()`(3RT)
`sema_destroy()`	`sem_destroy()`(3RT)
`sema_wait()`	`sem_wait()`(3RT)
`sema_post()`	`sem_post()`(3RT)
`sema_trywait()`	`sem_trywait()`(3RT)
`fork1()`	`fork()`
-	`pthread_atfork()`
`forkall()`, multiple thread copy	-
-	`pthread_mutexattr_init()`
-	`pthread_mutexattr_destroy()`
type argument in `mutex_init()`	`pthread_mutexattr_setpshared()`
-	`pthread_mutexattr_getpshared()`
-	`pthread_mutex_attr_settype()`
-	`pthread_mutex_attr_gettype()`
-	`pthread_condattr_init()`
-	`pthread_condattr_destroy()`
type argument in `cond_init()`	`pthread_condattr_setpshared()`
-	`pthread_condattr_getpshared()`

TABLE 6–2 Oracle Solaris Threads and POSIX pthreads Comparison *(Continued)*

Oracle Solaris Threads	pthreads
-	`pthread_attr_init()`
-	`pthread_attr_destroy()`
`THR_BOUND` flag in `thr_create()`	`pthread_attr_setscope()`
-	`pthread_attr_getscope()`
-	`pthread_attr_setguardsize()`
-	`pthread_attr_getguardsize()`
`stack_size` argument in `thr_create()`	`pthread_attr_setstacksize()`
-	`pthread_attr_getstacksize()`
`stack_addr` argument in `thr_create()`	`pthread_attr_setstack()`
-	`pthread_attr_getstack()`
`THR_DETACH` flag in `thr_create()`	`pthread_attr_setdetachstate()`
-	`pthread_attr_getdetachstate()`
-	`pthread_attr_setschedparam()`
-	`pthread_attr_getschedparam()`
-	`pthread_attr_setinheritsched()`
-	`pthread_attr_getinheritsched()`
-	`pthread_attr_setsschedpolicy()`
-	`pthread_attr_getschedpolicy()`

To use the Oracle Solaris threads functions described in this chapter for Solaris 9 and previous releases, you must link with the Oracle Solaris threads library `-lthread`.

Operation is virtually the same for both Oracle Solaris threads and for pthreads, even though the function names or arguments might differ. Only a brief example consisting of the correct include file and the function prototype is presented. Where return values are not given for the Oracle Solaris threads functions, see the appropriate pages in *man pages section 3: Basic Library Functions* for the function return values.

For more information on Oracle Solaris related functions, see the related pthreads documentation for the similarly named function.

Where Oracle Solaris threads functions offer capabilities that are not available in pthreads, a full description of the functions is provided.

Unique Oracle Solaris Threads Functions

This section describes unique Oracle Solaris threads functions: suspending thread execution and continuing a suspended thread.

Suspending Thread Execution

`thr_suspend`(3C) immediately suspends the execution of the thread specified by *target_thread*. On successful return from `thr_suspend()`, the suspended thread is no longer executing.

Because `thr_suspend()` suspends the target thread with no regard to the locks that the thread might be holding, you must use `thr_suspend()` with extreme care. If the suspending thread calls a function that requires a lock held by the suspended target thread, deadlock will result.

thr_suspend Syntax

```
#include <thread.h>

int thr_suspend(thread_t tid);
```

After a thread is suspended, subsequent calls to `thr_suspend()` have no effect. Signals cannot awaken the suspended thread. The signals remain pending until the thread resumes execution.

In the following synopsis, `pthread_t` *tid* as defined in pthreads is the same as `thread_t` *tid* in Oracle Solaris threads. *tid* values can be used interchangeably either by assignment or through the use of casts.

```
thread_t tid; /* tid from thr_create() */

/* pthreads equivalent of Solaris tid from thread created */
/* with pthread_create() */
pthread_t ptid;

int ret;

ret = thr_suspend(tid);

/* using pthreads ID variable with a cast */
ret = thr_suspend((thread_t) ptid);
```

thr_suspend Return Values

`thr_suspend()` returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, `thr_suspend()` fails and returns the corresponding value.

`ESRCH`

Description: *tid* cannot be found in the current process.

Continuing a Suspended Thread

thr_continue(3C) resumes the execution of a suspended thread. Once a suspended thread is continued, subsequent calls to thr_continue() have no effect.

thr_continue Syntax

```
#include <thread.h>

int thr_continue(thread_t tid);
```

A suspended thread is not awakened by a signal. The signal remains pending until the execution of the thread is resumed by thr_continue() .

pthread_t *tid* as defined in pthreads is the same as thread_t *tid* in Oracle Solaris threads. *tid* values can be used interchangeably either by assignment or through the use of casts.

```
thread_t tid; /* tid from thr_create()*/

/* pthreads equivalent of Oracle Solaris tid from thread created */
/* with pthread_create()*/
pthread_t ptid;

int ret;

ret = thr_continue(tid);

/* using pthreads ID variable with a cast */
ret = thr_continue((thread_t) ptid)
```

thr_continue Return Values

thr_continue() returns zero after completing successfully. Any other return value indicates that an error occurred. When the following condition occurs, thr_continue() fails and returns the corresponding value.

ESRCH

Description: *tid* cannot be found in the current process.

Similar Synchronization Functions: Read-Write Locks

Read-write locks allow simultaneous read access by many threads while restricting write access to only one thread at a time. This section discusses the following topics:

- Initializing a readers/writer lock
- Acquiring a read lock
- Trying to acquire a read lock
- Acquiring a write lock

- Trying to acquire a write
- Unlocking a readers/writer lock
- Destroying readers/writer lock state

When any thread holds the lock for reading, other threads can also acquire the lock for reading but must wait to acquire the lock for writing. If one thread holds the lock for writing, or is waiting to acquire the lock for writing, other threads must wait to acquire the lock for either reading or writing.

Read-write locks are slower than mutexes. But read-write locks can improve performance when the locks protect data not frequently written but are read by many concurrent threads.

Use read-write locks to synchronize threads in this process and other processes. Allocate read-write locks in memory that is writable and shared among the cooperating processes. See themmap(2) man page for information about mapping read-write locks for this behavior.

By default, the acquisition order is not defined when multiple threads are waiting for a read-write lock. However, to avoid writer starvation, the Oracle Solaris threads package tends to favor writers over readers of equal priority.

Read-write locks must be initialized before use.

Initialize a Read-Write Lock

Use rwlock_init(3C) to initialize the read-write lock pointed to by *rwlp* and to set the lock state to unlocked.

rwlock_init Syntax

```
#include <synch.h>  (or #include <thread.h>)

int rwlock_init(rwlock_t *rwlp, int type, void *
arg);
```

type can be one of the following values:

- USYNC_PROCESS The read-write lock can be used to synchronize threads in this process and other processes. *arg* is ignored.
- USYNC_THREAD The read-write lock can be used to synchronize threads in this process only. *arg* is ignored.

Multiple threads must not initialize the same read-write lock simultaneously. Read-write locks can also be initialized by allocation in zeroed memory, in which case a type of USYNC_THREAD is assumed. A read-write lock must not be reinitialized while other threads might be using the lock.

For POSIX threads, see "pthread_rwlock_init Syntax" on page 130 .

Initializing Read-Write Locks With Intraprocess Scope

```
#include <thread.h>
rwlock_t rwlp;
int ret;
/* to be used within this process only */
ret = rwlock_init(&rwlp, USYNC_THREAD, 0);
```

Initializing Read-Write Locks With Interprocess Scope

```
#include <thread.h>
rwlock_t rwlp;
int ret;
/* to be used among all processes */
ret = rwlock_init(&rwlp, USYNC_PROCESS, 0);
```

rwlock_init Return Values

`rwlock_init()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Invalid argument.

`EFAULT`

Description: *rwlp* or *arg* points to an illegal address.

Acquiring a Read Lock

Use `rw_rdlock`(3C) to acquire a read lock on the read-write lock pointed to by *rwlp*.

rw_rdlock Syntax

`#include <synch.h>` *(or* `#include <thread.h>`*)*

```
int rw_rdlock(rwlock_t *rwlp);
```

When the read-write lock is already locked for writing, the calling thread blocks until the write lock is released. Otherwise, the read lock is acquired. For POSIX threads, see "`pthread_rwlock_rdlock` Syntax" on page 130.

rw_rdlock Return Values

`rw_rdlock()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`
Description: Invalid argument.

`EFAULT`
Description: *rwlp* points to an illegal address.

Trying to Acquire a Read Lock

Use `rw_tryrdlock`(3C) to attempt to acquire a read lock on the read-write lock pointed to by *rwlp*.

rw_tryrdlock Syntax

```
#include <synch.h>  (or #include <thread.h>)

int rw_tryrdlock(rwlock_t *rwlp);
```

When the read-write lock is already locked for writing, `rw_tryrdlock()` returns an error. Otherwise, the read lock is acquired. For POSIX threads, see "`pthread_rwlock_tryrdlock` Syntax" on page 132.

rw_tryrdlock Return Values

`rw_tryrdlock()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`
Description: Invalid argument.

`EFAULT`
Description: *rwlp* points to an illegal address.

`EBUSY`
Description: The read-write lock pointed to by *rwlp* was already locked.

Acquiring a Write Lock

Use `rw_wrlock`(3C) to acquire a write lock on the read-write lock pointed to by *rwlp*.

rw_wrlock Syntax

```
#include <synch.h>  (or #include <thread.h>)

int rw_wrlock(rwlock_t *rwlp);
```

When the read-write lock is already locked for reading or writing, the calling thread blocks until all read locks and write locks are released. Only one thread at a time can hold a write lock on a read-write lock. For POSIX threads, see "`pthread_rwlock_wrlock` Syntax" on page 133.

rw_wrlock Return Values

`rw_wrlock()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Invalid argument.

`EFAULT`

Description: *rwlp* points to an illegal address.

Trying to Acquire a Write Lock

Use `rw_trywrlock(3C)` to attempt to acquire a write lock on the read-write lock pointed to by *rwlp*.

rw_trywrlock Syntax

```
#include <synch.h>  (or #include <thread.h>)

int rw_trywrlock(rwlock_t *rwlp);
```

When the read-write lock is already locked for reading or writing, `rw_trywrlock()` returns an error. For POSIX threads, see "`pthread_rwlock_trywrlock` Syntax" on page 134.

rw_trywrlock Return Values

`rw_trywrlock()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Invalid argument.

`EFAULT`

Description: *rwlp* points to an illegal address.

`EBUSY`

Description: The read-write lock pointed to by *rwlp* was already locked.

Unlock a Read-Write Lock

Use `rw_unlock(3C)` to unlock a read-write lock pointed to by *rwlp*.

rw_unlock Syntax

```
#include <synch.h>  (or #include <thread.h>)

int rw_unlock(rwlock_t *rwlp);
```

The read-write lock must be locked, and the calling thread must hold the lock either for reading or writing. When any other threads are waiting for the read-write lock to become available, one of the threads is unblocked. For POSIX threads, see “`pthread_rwlock_unlock` Syntax” on page 135.

rw_unlock Return Values

`rw_unlock()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Invalid argument.

`EFAULT`

Description: *rwlp* points to an illegal address.

Destroying the Read-Write Lock State

Use `rwlock_destroy(3C)` to destroy any state that is associated with the read-write lock pointed to by *rlwp*.

rwlock_destroy Syntax

```
#include <synch.h>  (or #include <thread.h>)

int rwlock_destroy(rwlock_t *rwlp);
```

The space for storing the read-write lock is not freed. For POSIX threads, see “`pthread_rwlock_destroy` Syntax” on page 136.

Example 6–1 uses a bank account to demonstrate read-write locks. While the program could allow multiple threads to have concurrent read-only access to the account balance, only a single writer is allowed. Note that the `get_balance()` function needs the lock to ensure that the addition of the checking and saving balances occurs atomically.

EXAMPLE 6–1 Read-Write Bank Account

```
rwlock_t account_lock;
float checking_balance = 100.0;
float saving_balance = 100.0;
...
rwlock_init(&account_lock, 0, NULL);
...

float
get_balance() {
    float bal;

    rw_rdlock(&account_lock);
    bal = checking_balance + saving_balance;
    rw_unlock(&account_lock);
    return(bal);
}

void
transfer_checking_to_savings(float amount) {
    rw_wrlock(&account_lock);
    checking_balance = checking_balance - amount;
    saving_balance = saving_balance + amount;
    rw_unlock(&account_lock);
}
```

rwlock_destroy Return Values

`rwlock_destroy()` returns zero after completing successfully. Any other return value indicates that an error occurred. When any of the following conditions occurs, the function fails and returns the corresponding value.

`EINVAL`

Description: Invalid argument.

`EFAULT`

Description: *rwlp* points to an illegal address.

Similar Oracle Solaris Threads Functions

TABLE 6–3 Similar Oracle Solaris Threads Functions

Operation	Related Function Description
Create a thread	"`thr_create` Syntax" on page 180
Get the minimal stack size	"`thr_min_stack` Syntax" on page 182
Get the thread identifier	"`thr_self` Syntax" on page 183
Yield thread execution	"`thr_yield` Syntax" on page 183

TABLE 6–3 Similar Oracle Solaris Threads Functions (Continued)

Operation	Related Function Description
Send a signal to a thread	"`thr_kill` Syntax" on page 184
Access the signal mask of the calling thread	"`thr_sigsetmask` Syntax" on page 184
Terminate a thread	"`thr_exit` Syntax" on page 185
Wait for thread termination	"`thr_join` Syntax" on page 185
Create a thread-specific data key	"`thr_keycreate` Syntax" on page 187
Set thread-specific data	"`thr_setspecific` Syntax" on page 188
Get thread-specific data	"`thr_getspecific` Syntax" on page 188
Set the thread priority	"`thr_setprio` Syntax" on page 189
Get the thread priority	"`thr_getprio` Syntax" on page 190

Creating a Thread

The `thr_create(3C)` routine is one of the most elaborate of all routines in the Oracle Solaris threads interface.

Use `thr_create(3C)` to add a new thread of control to the current process. For POSIX threads, see "`pthread_create` Syntax" on page 29.

thr_create Syntax

```
#include <thread.h>

int thr_create(void *stack_base, size_t stack_size,
          void *(*start_routine) (void *), void *arg,
          long flags,
          thread_t *new_thread);

size_t thr_min_stack(void);
```

Note that the new thread does not inherit pending signals, but the thread does inherit priority and signal masks.

stack_base. Contains the address for the stack that the new thread uses. If *stack_base* is `NULL`, then `thr_create()` allocates a stack for the new thread with at least *stack_size* bytes.

stack_size. Contains the size, in number of bytes, for the stack that the new thread uses. If *stack_size* is zero, a default size is used. In most cases, a zero value works best. If *stack_size* is not zero, *stack_size* must be greater than the value returned by `thr_min_stack()`.

In general, you do not need to allocate stack space for threads. The system allocates 1 megabyte of virtual memory for each thread's stack with no reserved swap space. The system uses the `-MAP_NORESERVE` option of `mmap(2)` to make the allocations.

start_routine. Contains the function with which the new thread begins execution. When `start_routine()` returns, the thread exits with the exit status set to the value returned by *start_routine* . See "`thr_exit` Syntax" on page 185.

arg. Can be any variable described by `void` , which is typically any 4-byte value. Any larger value must be passed indirectly by having the argument point to the variable.

Note that you can supply only one argument. To get your procedure to take multiple arguments, encode the multiple arguments as a single argument, such as by putting the arguments in a structure.

flags. Specifies attributes for the created thread. In most cases a zero value works best.

The value in *flags* is constructed from the bitwise inclusive OR of the following arguments:

- `THR_SUSPENDED`. Suspends the new thread, and does not execute *start_routine* until the thread is started by `thr_continue()`. Use `THR_SUSPENDED` to operate on the thread, such as changing its priority, before you run the thread.
- `THR_DETACHED`. Detaches the new thread so that its thread ID and other resources can be reused as soon as the thread terminates. Set `THR_DETACHED` when you do not want to wait for the thread to terminate.

Note – When no explicit synchronization is allocated, an unsuspended, detached thread can fail. On failure, the thread ID is reassigned to another new thread before its creator returns from `thr_create()`.

- `THR_BOUND`. Permanently binds the new thread to an LWP. The new thread is a *bound thread*. Starting with the Solaris 9 release, no distinction is made by the system between bound and unbound threads. All threads are treated as bound threads.
- `THR_DAEMON`. Marks the new thread as a daemon. A daemon thread is always detached. `THR_DAEMON` implies `THR_DETACHED`. The process exits when all nondaemon threads exit. Daemon threads do not affect the process exit status and are ignored when counting the number of thread exits.

  A process can exit either by calling `exit()` or by having every thread in the process that was not created with the `THR_DAEMON` flag call `thr_exit(3C)`. An application or a library that the process calls can create one or more threads that should be ignored (not counted) in the decision of whether to exit. The `THR_DAEMON` flag identifies threads that are not counted in the process exit criterion.

new_thread. When *new_thread* is not `NULL`, it points to where the ID of the new thread is stored when `thr_create()` is successful. The caller is responsible for supplying the storage pointed to by this argument. The ID is valid only within the calling process.

If you are not interested in this identifier, supply a NULL value to *new_thread*.

thr_create Return Values

`thr_create()` returns zero when the function completes successfully. Any other return value indicates that an error occurred. When any of the following conditions is detected, `thr_create()` fails and returns the corresponding value.

`EAGAIN`

Description: A system limit is exceeded, such as when too many LWPs have been created.

`ENOMEM`

Description: Insufficient memory was available to create the new thread.

`EINVAL`

Description: *stack_base* is not `NULL` and *stack_size* is less than the value returned by `thr_min_stack()`.

Getting the Minimal Stack Size

Use `thr_min_stack`(3C) to get the minimum stack size for a thread.

Stack behavior in Oracle Solaris threads is generally the same as stack behavior in pthreads. For more information about stack setup and operation, see "About Stacks" on page 65.

thr_min_stack Syntax

```
#include <thread.h>
```

```
size_t thr_min_stack(void);
```

`thr_min_stack()` returns the amount of space that is needed to execute a null thread. A null thread is a thread that is created to execute a null procedure. Useful threads need more than the absolute minimum stack size, so be very careful when reducing the stack size.

A thread that executes more than a null procedure should allocate a stack size that is larger than the size of `thr_min_stack()`.

When a thread is created with a user-supplied stack, the user must reserve enough space to run the thread. A dynamically linked execution environment increases the difficulty of determining the thread minimal stack requirements.

You can specify a custom stack in two ways. The first is to supply a `NULL` for the stack location, thereby asking the runtime library to allocate the space for the stack, but to supply the desired size in the stacksize parameter to `thr_create()`.

The other approach is to take overall aspects of stack management and supply a pointer to the stack to `thr_create()`. This means that you are responsible not only for stack allocation but also for stack deallocation. When the thread terminates, you must arrange for the disposal of the thread's stack.

When you allocate your own stack, be sure to append a red zone to its end by calling `mprotect(2)`.

Most users should not create threads with user-supplied stacks. User-supplied stacks exist only to support applications that require complete control over their execution environments.

Instead, users should let the system manage stack allocation. The system provides default stacks that should meet the requirements of any created thread.

thr_min_stack Return Values

No errors are defined.

Acquiring the Thread Identifier

Use `thr_self`(3C) to get the ID of the calling thread. For POSIX threads, see "`pthread_self` Syntax" on page 38.

thr_self Syntax

```
#include <thread.h>

thread_t thr_self(void);
```

thr_self Return Values

No errors are defined.

Yield Thread Execution

`thr_yield`(3C) causes the current thread to yield its execution in favor of another thread with the same or greater priority. Otherwise, `thr_yield()` has no effect. However, calling `thr_yield()` does not guarantee that the thread yields its execution.

thr_yield Syntax

```
#include <thread.h>

void thr_yield(void);
```

thr_yield Return Values

`thr_yield()` returns nothing and does not set `errno` .

Send a Signal to a Thread

`thr_kill(3C)` sends a signal to a thread. For POSIX threads, see "`pthread_kill` Syntax" on page 43.

thr_kill Syntax

```
#include <thread.h>
#include <signal.h>
int thr_kill(thread_t target_thread, int sig);
```

thr_kill Return Values

Upon successful completion, `thr_kill()` returns 0. When any of the following conditions is detected, `thr_kill()` fails and returns the corresponding value. When a failure occurs, no signal is sent.

`ESRCH`

Description: No thread was found associated with the thread designated by *thread* ID.

`EINVAL`

Description: The *sig* argument value is not zero. *sig* is an invalid or unsupported signal number.

Access the Signal Mask of the Calling Thread

Use `thr_sigsetmask(3C)` to change or examine the signal mask of the calling thread.

thr_sigsetmask Syntax

```
#include <thread.h>
#include <signal.h>
int thr_sigsetmask(int how, const sigset_t *set,
          sigset_t *oset);
```

`thr_sigsetmask()` changes or examines a calling thread's signal mask. Each thread has its own signal mask. A new thread inherits the calling thread's signal mask and priority. However, pending signals are not inherited. Pending signals for a new thread will be empty.

If the value of the argument *set* is not NULL, *set* points to a set of signals that can modify the currently blocked set. If the value of *set* is NULL, the value of *how* is insignificant and the thread's signal mask is unmodified. Use this behavior to inquire about the currently blocked signals.

The value of *how* specifies the method in which the set is changed. *how* takes one of the following values.

- SIG_BLOCK. *set* corresponds to a set of signals to block. The signals are added to the current signal mask.
- SIG_UNBLOCK. *set* corresponds to a set of signals to unblock. These signals are deleted from the current signal mask.
- SIG_SETMASK. *set* corresponds to the new signal mask. The current signal mask is replaced by *set*.

thr_sigsetmask Return Values

Upon successful completion, `thr_sigsetmask()` returns 0. When any of the following conditions is detected, `thr_sigsetmask()` fails and returns the corresponding value.

EINVAL

Description: *set* is not NULL and the value of *how* is not defined.

Terminate a Thread

Use `thr_exit`(3C) to terminate a thread. For POSIX threads, see "`pthread_exit` Syntax" on page 45.

thr_exit Syntax

```
#include <thread.h>

void thr_exit(void *status);
```

thr_exit Return Values

`thr_exit()` does not return to its caller.

Wait for Thread Termination

Use `thr_join`(3C) to wait for a target thread to terminate. For POSIX threads, see "`pthread_join` Syntax" on page 30.

thr_join Syntax

```
#include <thread.h>

int thr_join(thread_t tid, thread_t *departedid, void **status);
```

The target thread must be a member of the current process. The target thread cannot be a detached thread or a daemon thread.

Several threads cannot wait for the same thread to complete. One thread will complete successfully. The others will terminate with an ESRCH error.

thr_join() will not block processing of the calling thread if the target thread has already terminated.

thr_join, Join Specific

```
#include <thread.h>

thread_t tid;
thread_t departedid;
int ret;
void *status;

/* waiting to join thread "tid" with status */
ret = thr_join(tid, &departedid, &status);

/* waiting to join thread "tid" without status */
ret = thr_join(tid, &departedid, NULL);

/* waiting to join thread "tid" without return id and status */
ret = thr_join(tid, NULL, NULL);
```

When the *tid* is (thread_t)0, then thread_join() waits for any undetached thread in the process to terminate. In other words, when no thread identifier is specified, any undetached thread that exits causes thread_join() to return.

thr_join, Join Any

```
#include <thread.h>

thread_t tid;
thread_t departedid;
int ret;
void *status;

/* waiting to join any non-detached thread with status */
ret = thr_join(0, &departedid, &status);
```

By indicating 0 as the thread ID in the Oracle Solaris thr_join(), a join takes place when any non detached thread in the process exits. The *departedid* indicates the thread ID of the exiting thread.

thr_join Return Values

thr_join() returns 0 if successful. When any of the following conditions is detected, thr_join() fails and returns the corresponding value.

ESRCH

Description: No undetached thread is found which corresponds to the target thread ID.

EDEADLK

Description: A deadlock was detected or the value of the target thread specifies the calling thread.

Creating a Thread-Specific Data Key

thr_keycreate(3C) allocates a key that is used to identify thread-specific data in a process. The key is global to all threads in the process. Each thread binds a value to the key when the key gets created.

Except for the function names and arguments, thread-specific data is the same for Oracle Solaris threads as thread-specific data is for POSIX threads. The synopses for the Oracle Solaris functions are described in this section. For POSIX threads, see "pthread_key_create Syntax" on page 33.

thr_keycreate Syntax

```
#include <thread.h>

int thr_keycreate(thread_key_t *keyp,
    void (*destructor) (void *value));
```

keyp independently maintains specific values for each binding thread. Each thread is initially bound to a private element of *keyp* that allows access to its thread-specific data. Upon key creation, a new key is assigned the value NULL for all active threads. Additionally, upon thread creation, all previously created keys in the new thread are assigned the value NULL.

An optional *destructor* function can be associated with each *keyp*. Upon thread exit, if a *keyp* has a non-NULL *destructor* and the thread has a non-NULL *value* associated with *keyp* , the *destructor* is called with the currently associated *value*. If more than one *destructor* exists for a thread when it exits, the order of destructor calls is unspecified.

thr_keycreate Return Values

thr_keycreate() returns 0 if successful. When any of the following conditions is detected, thr_keycreate() fails and returns the corresponding value.

EAGAIN

Description: The system does not have the resources to create another thread-specific data key, or the number of keys exceeds the per-process limit for PTHREAD_KEYS_MAX.

ENOMEM

Description: Insufficient memory is available to associate *value* with *keyp*.

Setting the Thread-Specific Data Value

thr_setspecific(3C) binds *value* to the thread-specific data key, *key*, for the calling thread. For POSIX threads, see "pthread_setspecific Syntax" on page 35.

thr_setspecific Syntax

```
#include <thread.h>

int thr_setspecific(thread_key_t key, void *value);
```

thr_setspecific Return Values

`thr_setspecific()` returns 0 if successful. When any of the following conditions is detected, `thr_setspecific()` fails and returns the corresponding value.

`ENOMEM`

Description: Insufficient memory is available to associate *value* with *keyp*.

`EINVAL`

Description: *keyp* is invalid.

Getting the Thread-Specific Data Value

`thr_getspecific`(3C) stores the current value bound to *key* for the calling thread into the location pointed to by *valuep*. For POSIX threads, see "`pthread_getspecific` Syntax" on page 36.

thr_getspecific Syntax

```
#include <thread.h>

int thr_getspecific(thread_key_t key, void **valuep);
```

thr_getspecific Return Values

`thr_getspecific()` returns 0 if successful. When any of the following conditions is detected, `thr_getspecific()` fails and returns the corresponding value.

`ENOMEM`

Description: Insufficient memory is available to associate *value* with *keyp*.

`EINVAL`

Description: *keyp* is invalid.

Set the Thread Priority

In Oracle Solaris threads, a thread created with a priority other than the priority of its parents is created in SUSPEND mode. While suspended, the thread's priority is modified using the `thr_setprio(3C)` function call. After `thr_setprio()` completes, the thread resumes execution.

A higher priority thread receives precedence over lower priority threads with respect to synchronization object contention.

thr_setprio Syntax

thr_setprio(3C) changes the priority of the thread, specified by *tid*, within the current process to the priority specified by *newprio*. For POSIX threads, see "pthread_setschedparam Syntax" on page 40.

```
#include <thread.h>

int thr_setprio(thread_t tid, int newprio)
```

The range of valid priorities for a thread depends on its scheduling policy.

```
thread_t tid;
int ret;
int newprio = 20;

/* suspended thread creation */
ret = thr_create(NULL, NULL, func, arg, THR_SUSPENDED, &tid);

/* set the new priority of suspended child thread */
ret = thr_setprio(tid, newprio);

/* suspended child thread starts executing with new priority */
ret = thr_continue(tid);
```

thr_setprio Return Values

thr_setprio() returns 0 if successful. When any of the following conditions is detected, thr_setprio() fails and returns the corresponding value.

ESRCH

Description: The value specified by *tid* does not refer to an existing thread.

EINVAL

Description: The value of *priority* is invalid for the scheduling policy of the specified thread.

EPERM

Description: The caller does not have the appropriate permission to set the priority to the value specified.

Get the Thread Priority

Use thr_getprio(3C) to get the current priority for the thread. Each thread inherits a priority from its creator. thr_getprio() stores the current priority, *tid*, in the location pointed to by *newprio*. For POSIX threads, see "pthread_getschedparam Syntax" on page 41.

thr_getprio Syntax

```
#include <thread.h>

int thr_getprio(thread_t tid, int *newprio)
```

thr_getprio Return Values

`thr_getprio()` returns 0 if successful. When the following condition is detected, `thr_getprio()` fails and returns the corresponding value.

`ESRCH`

Description: The value specified by *tid* does not refer to an existing thread.

Similar Synchronization Functions: Mutual Exclusion Locks

- Initializing a mutex
- Destroying a mutex
- Acquiring a mutex
- Releasing a mutex
- Trying to acquire a mutex

Initialize a Mutex

Use `mutex_init`(3C) to initialize the mutex pointed to by *mp*. For POSIX threads, see "Initializing a Mutex" on page 86.

mutex_init(3C) Syntax

```
#include <synch.h>
#include <thread.h>

int mutex_init(mutex_t *mp, int type, void *arg));
```

The *type* can be one of the following values.

- `USYNC_PROCESS`. The mutex can be used to synchronize threads in this process and other processes. *arg* is ignored.
- `USYNC_PROCESS_ROBUST`. The mutex can be used to *robustly* synchronize threads in this process and other processes. *arg* is ignored.
- `USYNC_THREAD`. The mutex can be used to synchronize threads in this process only. *arg* is ignored.

When a process fails while holding a `USYNC_PROCESS` lock, subsequent requestors of that lock hang. This behavior is a problem for systems that share locks with client processes because the client processes can be abnormally killed. To avoid the problem of hanging on a lock held by a dead process, use USYNC_PROCESS_ROBUST to lock the mutex. USYNC_PROCESS_ROBUST adds two capabilities:

- In the case of process death, all owned locks held by that process are unlocked.
- The next requestor for any of the locks owned by the failed process receives the lock. But, the lock is held with an error return indicating that the previous owner failed while holding the lock.

Mutexes can also be initialized by allocation in zeroed memory, in which case a *type* of USYNC_THREAD is assumed.

Multiple threads must not initialize the same mutex simultaneously. A mutex lock must not be reinitialized while other threads might be using the mutex.

Mutexes With Intraprocess Scope

```
#include <thread.h>

mutex_t mp;
int ret;

/* to be used within this process only */
ret = mutex_init(&mp, USYNC_THREAD, 0);
```

Mutexes With Interprocess Scope

```
#include <thread.h>

mutex_t mp;
int ret;

/* to be used among all processes */
ret = mutex_init(&mp, USYNC_PROCESS, 0);
```

Mutexes With Interprocess Scope-Robust

```
#include <thread.h>

mutex_t mp;
int ret;

/* to be used among all processes */
ret = mutex_init(&mp, USYNC_PROCESS_ROBUST, 0);
```

mutex_init Return Values

mutex_init() returns 0 if successful. When any of the following conditions is detected, mutex_init() fails and returns the corresponding value.

EFAULT
  Description: *mp* points to an illegal address.

EINVAL
  Description: The value specified by *mp* is invalid.

ENOMEM
  Description: System has insufficient memory to initialize the mutex.

EAGAIN
  Description: System has insufficient resources to initialize the mutex.

EBUSY
  Description: System detected an attempt to reinitialize an active mutex.

Destroy a Mutex

Use `mutex_destroy(3C)` to destroy any state that is associated with the mutex pointed to by *mp* . The space for storing the mutex is not freed. For POSIX threads, see "`pthread_mutex_destroy` Syntax" on page 93.

mutex_destroy Syntax

```
#include <thread.h>

int mutex_destroy (mutex_t *mp);
```

mutex_destroy Return Values

`mutex_destroy()` returns 0 if successful. When the following condition is detected, `mutex_destroy()` fails and returns the corresponding value.

EFAULT
  Description: *mp* points to an illegal address.

Acquiring a Mutex

Use `mutex_lock(3C)` to lock the mutex pointed to by *mp*. When the mutex is already locked, the calling thread blocks until the mutex becomes available. Blocked threads wait on a prioritized queue. For POSIX threads, see "`pthread_mutex_lock` Syntax" on page 88.

mutex_lock Syntax

```
#include <thread.h>

int mutex_lock(mutex_t *mp);
```

mutex_lock Return Values

`mutex_lock()` returns 0 if successful. When any of the following conditions is detected, `mutex_lock()` fails and returns the corresponding value.

`EFAULT`
: **Description:** *mp* points to an illegal address.

`EDEADLK`
: **Description:** The mutex is already locked and is owned by the calling thread.

Releasing a Mutex

Use `mutex_unlock(3C)` to unlock the mutex pointed to by *mp*. The mutex must be locked. The calling thread must be the thread that last locked the mutex, the owner. For POSIX threads, see "`pthread_mutex_unlock` Syntax" on page 89.

mutex_unlock Syntax

```
#include <thread.h>

int mutex_unlock(mutex_t *mp);
```

mutex_unlock Return Values

`mutex_unlock()` returns 0 if successful. When any of the following conditions is detected, `mutex_unlock()` fails and returns the corresponding value.

`EFAULT`
: **Description:** *mp* points to an illegal address.

`EPERM`
: **Description:** The calling thread does not own the mutex.

Trying to Acquire a Mutex

Use `mutex_trylock(3C)` to attempt to lock the mutex pointed to by *mp*. This function is a nonblocking version of `mutex_lock()`. For POSIX threads, see "`pthread_mutex_trylock` Syntax" on page 90.

mutex_trylock Syntax

```
#include <thread.h>

int mutex_trylock(mutex_t *mp);
```

mutex_trylock Return Values

`mutex_trylock()` returns 0 if successful. When any of the following conditions is detected, `mutex_trylock()` fails and returns the corresponding value.

`EFAULT`

Description: *mp* points to an illegal address.

`EBUSY`

Description: The system detected an attempt to reinitialize an active mutex.

Similar Synchronization Functions: Condition Variables

- Initializing a condition variable
- Destroying a condition variable
- Waiting for a condition
- Waiting for an absolute time
- Waiting for a time interval
- Unblocking one thread
- Unblocking all threads

Initialize a Condition Variable

Use `cond_init`(3C) to initialize the condition variable pointed to by *cv*.

cond_init Syntax

```
#include <thread.h>

int cond_init(cond_t *cv, int type, int arg);
```

The *type* can be one of the following values:

- `USYNC_PROCESS`. The condition variable can be used to synchronize threads in this process and other processes. *arg* is ignored.
- `USYNC_THREAD` The condition variable can be used to synchronize threads in this process only. *arg* is ignored.

Condition variables can also be initialized by allocation in zeroed memory, in which case a type of `USYNC_THREAD` is assumed.

Multiple threads must not initialize the same condition variable simultaneously. A condition variable must not be reinitialized while other threads might be using the condition variable.

For POSIX threads, see "`pthread_condattr_init` Syntax" on page 103 .

Condition Variables With Intraprocess Scope

```
#include <thread.h>

cond_t cv;
int ret;

/* to be used within this process only */
ret = cond_init(cv, USYNC_THREAD, 0);
```

Condition Variables With Interprocess Scope

```
#include <thread.h>

cond_t cv;
int ret;

/* to be used among all processes */
ret = cond_init(&cv, USYNC_PROCESS, 0);
```

cond_init Return Values

`cond_init()` returns 0 if successful. When any of the following conditions is detected, `cond_init()` fails and returns the corresponding value.

`EFAULT`

Description: *cv* points to an illegal address.

`EINVAL`

Description: *type* is not a recognized type.

Destroying a Condition Variable

Use `cond_destroy`(3C) to destroy state that is associated with the condition variable pointed to by *cv* . The space for storing the condition variable is not freed. For POSIX threads, see "`pthread_condattr_destroy` Syntax" on page 104.

cond_destroy Syntax

```
#include <thread.h>

int cond_destroy(cond_t *cv);
```

cond_destroy Return Values

`cond_destroy()` returns 0 if successful. When any of the following conditions is detected, `cond_destroy()` fails and returns the corresponding value.

EFAULT

Description: *cv* points to an illegal address.

EBUSY

Description: The system detected an attempt to destroy an active condition variable.

Waiting for a Condition

Use cond_wait(3C) to atomically release the mutex pointed to by *mp* and cause the calling thread to block on the condition variable pointed to by *cv*. The blocked thread can be awakened by cond_signal(), cond_broadcast(), or when interrupted by delivery of a signal or a fork().

cond_wait() always returns with the mutex locked and owned by the calling thread, even when returning an error.

cond_wait Syntax

```
#include <thread.h>

int cond_wait(cond_t *cv, mutex_t *mp);
```

cond_wait Return Values

cond_wait() returns 0 if successful. When any of the following conditions is detected, cond_wait() fails and returns the corresponding value.

EFAULT

Description: *cv* points to an illegal address.

EINTR

Description: The wait was interrupted by a signal.

Wait for an Absolute Time

cond_timedwait(3C) is very similar to cond_wait(), except that cond_timedwait() does not block past the time of day specified by *abstime* . For POSIX threads, see "pthread_cond_timedwait Syntax" on page 111.

cond_timedwait Syntax

```
#include <thread.h>

int cond_timedwait(cond_t *cv, mutex_t *mp, timestruct_t abstime);
```

cond_timedwait() always returns with the mutex locked and owned by the calling thread, even when returning an error.

The `cond_timedwait()` function blocks until the condition is signaled or until the time of day specified by the last argument has passed. The timeout is specified as the time of day so the condition can be retested efficiently without recomputing the time-out value.

cond_timedwait Return Values

`cond_timedwait()` returns 0 if successful. When any of the following conditions is detected, `cond_timedwait()` fails and returns the corresponding value.

`EFAULT`

Description: *cv* points to an illegal address.

`ETIME`

Description: The time specified by *abstime* has expired.

`EINVAL`

Description: *abstime* is invalid.

Waiting for a Time Interval

`cond_reltimedwait(3C)` is very similar to `cond_timedwait()`, except for the value for the third argument. `cond_reltimedwait()` takes a relative time interval value in its third argument rather than an absolute time of day value. For POSIX threads, see the `pthread_cond_reltimedwait_np(3C)` man page.

`cond_reltimedwait()` always returns with the mutex locked and owned by the calling thread even when returning an error. The `cond_reltimedwait()` function blocks until the condition is signaled or until the time interval specified by the last argument has elapsed.

cond_reltimedwait Syntax

```
#include <thread.h>

int cond_reltimedwait(cond_t *cv, mutex_t *mp,
    timestruct_t reltime);
```

cond_reltimedwait Return Values

`cond_reltimedwait()` returns 0 if successful. When any of the following conditions is detected, `cond_reltimedwait()` fails and returns the corresponding value.

`EFAULT`

Description: *cv* points to an illegal address.

`ETIME`

Description: The time specified by *reltime* has expired.

Unblock One Thread

Use `cond_signal`(3C) to unblock one thread that is blocked on the condition variable pointed to by *cv* . If no threads are blocked on the condition variable, `cond_signal()` has no effect.

cond_signal Syntax

```
#include <thread.h>

int cond_signal(cond_t *cv);
```

cond_signal Return Values

`cond_signal()` returns 0 if successful. When the following condition is detected, `cond_signal()` fails and returns the corresponding value.

`EFAULT`

Description: *cv* points to an illegal address.

Unblock All Threads

Use `cond_broadcast`(3C) to unblock all threads that are blocked on the condition variable pointed to by *cv*. When no threads are blocked on the condition variable, then `cond_broadcast()` has no effect.

cond_broadcast Syntax

```
#include <thread.h>

int cond_broadcast(cond_t *cv);
```

cond_broadcast Return Values

`cond_broadcast()` returns 0 if successful. When the following condition is detected, `cond_broadcast()` fails and returns the corresponding value.

`EFAULT`

Description: *cv* points to an illegal address.

Similar Synchronization Functions: Semaphores

Semaphore operations are the same in both the Oracle Solaris Operating Environment and the POSIX environment. The function name changed from `sema_` in the Oracle Solaris Operating Environment to `sem_` in pthreads. This section discusses the following topics:

- Initializing a semaphore
- Incrementing a semaphore
- Blocking on a semaphore count
- Decrementing a semaphore count
- Destroying the semaphore state

Initialize a Semaphore

Use `sema_init(3C)` to initialize the semaphore variable pointed to by *sp* by *count* amount.

sema_init Syntax

```
#include <thread.h>

int sema_init(sema_t *sp, unsigned int count, int type,
    void *arg);
```

type can be one of the following values:

- `USYNC_PROCESS`. The semaphore can be used to synchronize threads in this process and other processes. Only one process should initialize the semaphore. *arg* is ignored.
- `USYNC_THREAD`. The semaphore can be used to synchronize threads in this process, only. *arg* is ignored.

Multiple threads must not initialize the same semaphore simultaneously. A semaphore must not be reinitialized while other threads might be using the semaphore.

Semaphores With Intraprocess Scope

```
#include <thread.h>

sema_t sp;
int ret;
int count;
count = 4;

/* to be used within this process only */
ret = sema_init(&sp, count, USYNC_THREAD, 0);
```

Semaphores With Interprocess Scope

```
#include <thread.h>

sema_t sp;
```

```
int ret;
int count;
count = 4;

/* to be used among all the processes */
ret = sema_init (&sp, count, USYNC_PROCESS, 0);
```

sema_init Return Values

`sema_init()` returns 0 if successful. When any of the following conditions is detected, `sema_init()` fails and returns the corresponding value.

`EINVAL`

Description: *sp* refers to an invalid semaphore.

`EFAULT`

Description: Either *sp* or *arg* point to an illegal address.

Increment a Semaphore

Use `sema_post`(3C) to atomically increment the semaphore pointed to by *sp*. When any threads are blocked on the semaphore, one thread is unblocked.

sema_post Syntax

```
#include <thread.h>

int sema_post(sema_t *sp);
```

sema_post Return Values

`sema_post()` returns 0 if successful. When any of the following conditions is detected, `sema_post()` fails and returns the corresponding value.

`EINVAL`

Description: *sp* refers to an invalid semaphore.

`EFAULT`

Description: *sp* points to an illegal address.

`EOVERFLOW`

Description: The semaphore value pointed to by *sp* exceeds `SEM_VALUE_MAX`.

Block on a Semaphore Count

Use sema_wait(3C) to block the calling thread until the count in the semaphore pointed to by *sp* becomes greater than zero. When the count becomes greater than zero, atomically decrement the count.

sema_wait Syntax

```
#include <thread.h>

int sema_wait(sema_t *sp);
```

sema_wait Return Values

`sema_wait()` returns 0 if successful. When any of the following conditions is detected, `sema_wait()` fails and returns the corresponding value.

`EINVAL`

Description: *sp* refers to an invalid semaphore.

`EINTR`

Description: The wait was interrupted by a signal.

Decrement a Semaphore Count

Use sema_trywait(3C) to atomically decrement the count in the semaphore pointed to by *sp* when the count is greater than zero. This function is a nonblocking version of `sema_wait()`.

sema_trywait Syntax

```
#include <thread.h>

int sema_trywait(sema_t *sp);
```

sema_trywait Return Values

`sema_trywait()` returns 0 if successful. When any of the following conditions is detected, `sema_trywait()` fails and returns the corresponding value.

`EINVAL`

Description: *sp* refers to an invalid semaphore.

`EBUSY`

Description: The semaphore pointed to by *sp* has a zero count.

Destroy the Semaphore State

Use sema_destroy(3C) to destroy any state that is associated with the semaphore pointed to by *sp*. The space for storing the semaphore is not freed.

sema_destroy(3C) Syntax

```
#include <thread.h>

int sema_destroy(sema_t *sp);
```

sema_destroy(3C) Return Values

sema_destroy() returns 0 if successful. When the following condition is detected, sema_destroy() fails and returns the corresponding value.

EINVAL
    Description: *sp* refers to an invalid semaphore.

Synchronizing Across Process Boundaries

Each of the synchronization primitives can be set up to be used across process boundaries. This cross-boundary setup is done by ensuring that the synchronization variable is located in a shared memory segment and by calling the appropriate init routine with type set to USYNC_PROCESS.

If type is set to USYNC_PROCESS, then the operations on the synchronization variables work just as the variables do when *type* is USYNC_THREAD.

```
mutex_init(&m, USYNC_PROCESS, 0);
rwlock_init(&rw, USYNC_PROCESS, 0);
cond_init(&cv, USYNC_PROCESS, 0);
sema_init(&s, count, USYNC_PROCESS, 0);
```

Example of Producer and Consumer Problem

Example 6–2 shows the producer and consumer problem with the producer and consumer in separate processes. The main routine maps zero-filled memory that main shares with its child process, into its address space. Note that mutex_init() and cond_init() must be called because the type of the synchronization variables is USYNC_PROCESS.

A child process is created to run the consumer. The parent runs the producer.

This example also shows the drivers for the producer and consumer. The producer_driver reads characters from stdin and calls the producer. The consumer_driver gets characters by calling the consumer and writes them to stdout.

The data structure for Example 6–2 is the same as that used for the solution with condition variables. See "Examples of Using Nested Locking With a Singly-Linked List" on page 96 .

EXAMPLE 6–2 Producer and Consumer Problem Using USYNC_PROCESS

```
main() {
    int zfd;
    buffer_t *buffer;

    zfd = open("/dev/zero", O_RDWR);
    buffer = (buffer_t *)mmap(NULL, sizeof(buffer_t),
        PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);
    buffer->occupied = buffer->nextin = buffer->nextout = 0;

    mutex_init(&buffer->lock, USYNC_PROCESS, 0);
    cond_init(&buffer->less, USYNC_PROCESS, 0);
    cond_init(&buffer->more, USYNC_PROCESS, 0);
    if (fork() == 0)
        consumer_driver(buffer);
    else
        producer_driver(buffer);
}

void producer_driver(buffer_t *b) {
    int item;

    while (1) {
        item = getchar();
        if (item == EOF) {
            producer(b, '\0');
            break;
        } else
            producer(b, (char)item);
    }
}

void consumer_driver(buffer_t *b) {
    char item;

    while (1) {
        if ((item = consumer(b)) == '\0')
            break;
        putchar(item);
    }
}
```

A child process is created to run the consumer. The parent runs the producer.

Special Issues for `fork()` and Oracle Solaris Threads

Prior to the Oracle Solaris 10 release, Oracle Solaris threads and POSIX threads defined the behavior of `fork()` differently. See "Process Creation: `exec` and `exit` Issues" on page 149 for a thorough discussion of `fork()` issues.

Oracle Solaris `libthread` supported both `fork()` and `fork1()`. The `fork()` call has "fork-all" semantics. `fork()` duplicated everything in the process, including threads and LWPs, creating a true clone of the parent. The `fork1()` call created a clone that had only one thread. The process state and address space are duplicated, but only the calling thread was cloned.

POSIX `libpthread` supported only `fork()`, which has the same semantics as `fork1()` in Oracle Solaris threads.

Whether `fork()` has "fork-all" semantics or "fork-one" semantics was dependent on which library is used. Link with `-lthread` to assign "fork-all" semantics to `fork()`. Link with `-lpthread` to assign "fork-one" semantics to `fork()`.

Effective with the Oracle Solaris 10 release, `fork()` has the same semantics in both Oracle Solaris threads and POSIX threads. More specifically, `fork1()` semantics replicate only the caller. A new function, `forkall()`, is provided for those applications that require replicate-all semantics.

See "Compiling and Linking a Multithreaded Program" on page 213 for more details.

CHAPTER 7

Safe and Unsafe Interfaces

This chapter defines MT-safety levels for functions and libraries. This chapter discusses the following topics:

- "Thread Safety" on page 205
- "MT Interface Safety Levels" on page 206
- "Async-Signal-Safe Functions in Oracle Solaris Threads" on page 208
- "MT Safety Levels for Libraries" on page 209

Thread Safety

Thread safety is the avoidance of data races. Data races occur when data are set to either correct or incorrect values, depending upon the order in which multiple threads access and modify the data.

When no sharing is intended, give each thread a private copy of the data. When sharing is important, provide explicit synchronization to make certain that the program behaves in a deterministic manner.

A procedure is thread safe when the procedure is logically correct when executed simultaneously by several threads. At a practical level, safety falls into the following recognized levels.

- Unsafe
- Thread safe, Serializable
- Thread safe, MT-Safe

An unsafe procedure can be made thread safe and able to be serialized by surrounding the procedure with statements to lock and unlock a mutex. Example 7–1 shows three simplified implementations of `fputs()`, initially thread unsafe.

Next is a serializable version of this routine with a single mutex protecting the procedure from concurrent execution problems. Actually, the single mutex is stronger synchronization than is usually necessary. When two threads are sending output to different files by using `fputs()`, one thread need not wait for the other thread. The threads need synchronization only when sharing an output file.

The last version is MT-safe. This version uses one lock for each file, allowing two threads to print to different files at the same time. So, a routine is MT-safe when the routine is thread safe, and the routine's execution does not negatively affect performance.

EXAMPLE 7–1 Degrees of Thread Safety

```
/* not thread-safe */
fputs(const char *s, FILE *stream) {
    char *p;
    for (p=s; *p; p++)
        putc((int)*p, stream);
}

/* serializable */
fputs(const char *s, FILE *stream) {
    static mutex_t mut;
    char *p;
    mutex_lock(&m);
    for (p=s; *p; p++)
        putc((int)*p, stream);

    mutex_unlock(&m);
}

/* MT-Safe */
mutex_t m[NFILE];
fputs(const char *s, FILE *stream) {
    char *p;
    mutex_lock(&m[fileno(stream)]);
    for (p=s; *p; p++)
        putc((int)*p, stream);
    mutex_unlock(&m[fileno(stream)]0;
}
```

MT Interface Safety Levels

The man pages for functions and interfaces indicate how well the function or interface supports threads. The ATTRIBUTES section of each man page lists the MT-Level attribute, which is set to one of the safety level categories listed in Table 7–1. These categories are explained more fully in the `attributes(5)` man page.

If a man page does not state explicitly that a function is MT-Safe, you must assume that the function is unsafe.

TABLE 7–1 Interface Safety Levels

Category	Description
Safe	This code can be called from a multithreaded application
Safe with exceptions	See the NOTES sections of the man page for a description of the exceptions.
Unsafe	This interface is not safe to use with multithreaded applications unless the application arranges for only one thread at a time to execute within the library.
MT-Safe	This interface is fully prepared for multithreaded access. The interface is both *safe* and supports some concurrency.
MT-Safe with exceptions	See the NOTES sections of the man page for a description of the exceptions.
Async-Signal-Safe	This routine can safely be called from a signal handler. A thread that is executing an Async-Signal-Safe routine does not deadlock with itself when interrupted by a signal. See "Async-Signal-Safe Functions in Oracle Solaris Threads" on page 208
Fork1–Safe	This interface releases locks it has held whenever Oracle Solaris `fork1`(2) or POSIX `fork`(2) is called.

Some functions have purposely not been made safe for the following reasons.

- The interface made MT-Safe would have negatively affected the performance of single-threaded applications.
- The library has an unsafe interface. For example, a function might return a pointer to a buffer in the stack. You can use re-entrant counterparts for some of these functions. The re-entrant function name is the original function name with "`_r`" appended.

Note – The only way to be certain that a function with a name not ending in "`_r`" is MT-Safe is to check the function's manual page. Use of a function identified as not MT-Safe must be protected by a synchronizing device or by restriction to the initial thread.

Reentrant Functions for Unsafe Interfaces

For most functions with unsafe interfaces, an MT-Safe version of the routine exists. The name of the MT-Safe routine is the name of the Unsafe routine with "`_r`" appended. For example, the MT-Safe version of `asctime()` is `asctime_r()`. The Table 7–2 "`_r`" routines are supplied in the Oracle Solaris environment.

TABLE 7–2 Reentrant Functions

asctime_r(3c)	gethostbyname_r(3nsl)	getservbyname_r(3socket)
ctermid_r(3c)	gethostent_r(3nsl)	getservbyport_r(3socket)
ctime_r(3c)	getlogin_r(3c)	getservent_r(3socket)
fgetgrent_r(3c)	getnetbyaddr_r(3socket)	getspent_r(3c)
fgetpwent_r(3c)	getnetbyname_r(3socket)	getspnam_r(3c)
fgetspent_r(3c)	getnetent_r(3socket)	gmtime_r(3c)
gamma_r(3m)	getnetgrent_r(3c)	lgamma_r(3m)
getauclassent_r(3bsm)	getprotobyname_r(3socket)	localtime_r(3c)
getauclassnam_r(3bsm)	getprotobynumber_r(3socket)	nis_sperror_r(3nsl)
getauevent_r(3bsm)	getprotoent_r(3socket)	rand_r(3c)
getauevnam_r(3bsm)	getpwent_r(3c)	readdir_r(3c)
getauevnum_r(3bsm)	getpwnam_r(3c)	strtok_r(3c)
getgrent_r(3c)	getpwuid_r(3c)	tmpnam_r(3c)
getgrgid_r(3c)	getrpcbyname_r(3nsl)	ttyname_r(3c)
getgrnam_r(3c)	getrpcbynumber_r(3nsl)	
gethostbyaddr_r(3nsl)	getrpcent_r(3nsl)	

Async-Signal-Safe Functions in Oracle Solaris Threads

Functions that can safely be called from signal handlers are *Async-Signal-Safe*. The IEEE Std 1003.1–2004 (POSIX) standard defines Async-Signal-Safe functions, which are listed in Table 5–2. In addition to these standard Async-Signal-Safe functions, the following functions from the Oracle Solaris threads interface are also Async-Signal-Safe:

- sema_post(3C)
- thr_sigsetmask(3C), similar to pthread_sigmask(3C)
- thr_kill(3C), similar to pthread_kill(3C)

MT Safety Levels for Libraries

All routines that can potentially be called by a thread from a multithreaded program should be MT-Safe. Therefore, two or more activations of a routine must be able to *correctly* execute concurrently. So, every library interface that a multithreaded program uses must be MT-Safe.

Not all libraries are now MT-Safe. The commonly used libraries that are MT-Safe are listed in the following table. The libraries are accessed in the `/usr/lib` directory.

TABLE 7–3 Some MT-Safe Libraries

Library	Comments
`libc`	Interfaces that are not safe have thread-safe interfaces of the form `*_r`, often with different semantics.
`libm`	Math library that is compliant with System V Interface Definition, Edition 3, X/Open, and ANSI C
`libmalloc`	Space-efficient memory allocation library, see `malloc`(3MALLOC)
`libmapmalloc`	Alternative `mmap`-based memory allocation library, see `mapmalloc`(3MALLOC)
`libnsl`	The TLI interface, XDR, RPC clients and servers, `netdir`, `netselect` and `getXXbyYY` interfaces are not safe, but have thread-safe interfaces of the form `getXXbyYY_r`
`libresolv`	Domain name server library routines
`libsocket`	Socket library for making network connections
`libX11`	X11 Windows library routines
`libCrun`	C++ runtime shared objects for Oracle C++ 5.0 compilers
`libCstd`	C++ standard library for Oracle C++ 5.0 compilers
`libiostream`	Classic `iostream` library for Oracle C++ 5.0 compilers
`libC.so.5`	C++ runtime and `iostream` library for Oracle C++ 4.0 compilers

Unsafe Libraries

Routines in libraries that are not guaranteed to be MT-Safe can safely be called by multithreaded programs only when such calls are single threaded.

CHAPTER 8

Compiling and Debugging

This chapter describes how to compile and debug multithreaded programs. This chapter discusses the following topics:

- "Setting Up the Oracle Solaris Environment for Developing Multithreaded Applications" on page 211
- "Compiling a Multithreaded Application" on page 211
- "Debugging a Multithreaded Program" on page 214

Setting Up the Oracle Solaris Environment for Developing Multithreaded Applications

To build software on the Oracle Solaris OS, you must install the tools you need on your development machine. Whether you want to use the standard tools that are bundled in Oracle Solaris OS, or use the Oracle Solaris Studio tools, you must first install the appropriate Oracle Solaris software for a developer environment.

Compiling a Multithreaded Application

This section explains how to compile a multithreaded program using the Oracle Solaris Studio C compiler. The Oracle Solaris Studio C compiler is optimized for parallel programming and includes many features that are not available in other C compilers. See the *Oracle Solaris Studio 12.3: C User's Guide* for more information about the C compiler.

Preparing for Compilation

Your application must include `<thread.h>` for Oracle Solaris threads and `<pthread.h>` for POSIX threads. You should include the appropriate file for the API you are using, or both files if your application uses both thread APIs. See the pthread.h(3HEAD) man page for more information. The application must also include <errno.h>, <limits.h>, <signal.h> , <unistd.h> files.

Choosing Oracle Solaris or POSIX Threads

The Oracle Solaris implementation of Pthreads is completely compatible with Oracle Solaris threads. You can use both Oracle Solaris threads and Pthreads in the same application. See the pthreads(5) man page for a discussion of the differences between the thread implementations. See also Chapter 6, "Programming With Oracle Solaris Threads," in this manual for information about differences.

One difference between the thread types is the behavior of the fork functions.

In the Solaris 9 release, the behavior of the fork() function depended on whether or not the application was linked with the POSIX threads library. When linked with -lthread (Oracle Solaris Threads) but not linked with -lpthread (POSIX Threads), fork() would duplicate in the child thread all the threads from the parent process. When the application was linked with -lpthread, whether or not also linked with -lthread, fork() was the same as fork1() and only the calling thread is duplicated.

Starting in the Oracle Solaris 10 release, a call to the forkall() function replicates in the child process all of the threads in the parent process. A call to fork1() replicates only the calling thread in the child process. In the Oracle Solaris 10 release, a call to fork() is identical to a call to fork1(); only the calling thread is replicated in the child process. This is the POSIX-specified behavior for fork(). Applications that require replicate-all fork semantics must call forkall().

Including <thread.h> or <pthread.h>

The include file <thread.h> contains declarations for the Oracle Solaris threads functions. To call any Oracle Solaris thread functions, your program needs to include <thread.h>. This file enables you to produce compiled code that is compatible with earlier releases of the Oracle Solaris software.

The include file <pthread.h> contains declarations for the Pthreads functions and is required if your program uses Pthreads.

You can mix Oracle Solaris threads and POSIX threads in the same application by including both <thread.h> and <pthread.h> in the application. Then when linking and compiling you need to specify the -lpthread flag to link in the pthread APIs.

When using -mt, the Oracle Solaris threads APIs will be linked automatically. Always use the -mt option instead of listing -lthread explicitly. To use Pthreads, specify the -mt option and -lpthread option on the link command line. The libpthread library provides an interface to libthread, so you still need libthread when using Pthreads.

Compiling and Linking a Multithreaded Program

The Oracle Solaris Studio C compiler (`cc`) provides the `-mt` option to compile and link multithreaded code. The `-mt` option assures that libraries are linked in appropriate order.

The `-mt` option must be used consistently. If you compile with `-mt` and link in a separate step, you must use the `-mt` option in the link step as well as the compile step. If you compile and link one translation unit with `-mt`, you must compile and link all units of the program with `-mt`.

Compiling and Linking in the POSIX Threads Environment

If your application uses only Pthreads or uses both Oracle Solaris threads and Pthreads, use the following command to compile and link:

`cc -mt` *[flag ...] file... [library...]* `-lpthread`

The `-mt` option links in the `libthread` library, while the `-lpthread` option links in the `libpthread` library. Both flags are needed when using Pthreads because `libpthread` provides an interface to `libthread`.

The `-mt` option can appear anywhere in the command line. The `-lpthread` option should come after any user libraries. The relative positions of `-mt` and `-lpthread` do not matter.

For example, the following lines are equivalent:

```
cc -mt -o myprog f1.o f2.o   -lmylib -lpthread
cc     -o myprog f1.o f2.o -mt -lmylib -lpthread
cc     -o myprog f1.o f2.o -lmylib -mt -lpthread
cc     -o myprog f1.o f2.o -lmylib -lpthread -mt
```

See the *Oracle Solaris Studio 12.3: C User's Guide* and the Oracle Solaris Studio 12.3 Command-Line Reference for more information about the `cc` command.

Compiling and Linking in the Oracle Solaris Threads Environment

In a Oracle Solaris threads environment, use the following options to compile and link your application:

If you application uses *only* Oracle Solaris threads, use the following command to compile and link:

`cc -mt` *[flag ...] file... [library...]*

The `-mt` option links in the `libthread` library.

See the*Oracle Solaris Studio 12.3: C User's Guide* for more information about the `cc` command.

Compiling and Linking in a Mixed Threads Environment

If your application uses both Pthreads and Oracle Solaris threads functions, you can compile and link with the same command used for compiling for Pthreads only:

```
cc -mt [ flag ... ] file... [ library... ] -lpthread
```

In mixed usage, you need to include both `thread.h` and `pthread.h`.

Linking With -lrt for POSIX Semaphores

The Oracle Solaris semaphore routines, `sema_*(3C)`, are contained in the standard C library. By contrast, you link with the `-lrt` library to get the standard `sem_*(3RT)` POSIX semaphore routines described in "Synchronization With Semaphores" on page 119.

Alternate Threads Library

The Solaris 8 release introduced an alternate threads library implementation that is located in the directories `/usr/lib/lwp` (32-bit) and `/usr/lib/lwp/64` (64-bit). In the Solaris 9 release, this implementation became the standard threads implementation found in `/usr/lib` and `/usr/lib/64`. Effective with the Oracle Solaris 10 release, all threads functionality has been moved into `libc` and no separate threads library is required. The `/usr/lib/lwp` directories are maintained for compatibility of Solaris 8 applications.

Debugging a Multithreaded Program

The following discussion describes characteristics that can cause bugs in multithreaded programs. Utilities that you can use to help debug your program are also described.

Common Oversights in Multithreaded Programs

The following list points out some of the more frequent oversights that can cause bugs in multithreaded programs.

- A pointer passed to the caller's stack as an argument to a new thread.
- The shared changeable state of global memory accessed without the protection of a synchronization mechanism leading to a *data race*. A data race occurs when two or more threads in a single process access the same memory location concurrently, and at least one of the threads tries to write to the location. When the threads do not use exclusive locks to control their accesses to that memory, the order of accesses is non-deterministic, and the computation may give different results from run to run depending on that order. Some data

races may be benign (for example, when the memory access is used for a busy-wait), but many data races are bugs in the program. The Thread Analyzer tool is useful for detecting data races. See "Detecting Data Races and Deadlocks Using Thread Analyzer" on page 216.

- Deadlocks caused by two threads trying to acquire rights to the same pair of global resources in alternate order. One thread controls the first resource and the other controls the second resource. Neither thread can proceed until the other gives up. The Thread Analyzer tool is also useful for detecting deadlocks. See "Detecting Data Races and Deadlocks Using Thread Analyzer" on page 216.
- Trying to reacquire a lock already held (recursive deadlock).
- Creating a hidden gap in synchronization protection. This gap in protection occurs when a protected code segment contains a function that frees and reacquires the synchronization mechanism before returning to the caller. The result is misleading. To the caller, the appearance is that the global data has been protected when the data actually has not been protected.
- When mixing UNIX signals with threads, and not using the `sigwait`(2) model for handling asynchronous signals.
- Calling `setjmp(3C)` and `longjmp(3C)`, and then long-jumping away without releasing the mutex locks.
- Failing to re-evaluate the conditions after returning from a call to `*_cond_wait()` or `*_cond_timedwait()`.
- Forgetting that default threads are created `PTHREAD_CREATE_JOINABLE` and must be reclaimed with `pthread_join(3C)`. Note that `pthread_exit(3C)` does not free up its storage space.
- Making deeply nested, recursive calls and using large automatic arrays can cause problems because multithreaded programs have a more limited stack size than single-threaded programs.
- Specifying an inadequate stack size, or using nondefault stacks.

Multithreaded programs, especially those containing bugs, often behave differently in two successive runs, even with identical inputs. This behavior is caused by differences in the order that threads are scheduled.

In general, multithreading bugs are statistical instead of deterministic. Tracing is usually a more effective method of finding the order of execution problems than is breakpoint-based debugging.

Tracing and Debugging with DTrace

DTrace is a comprehensive dynamic tracing facility that is built into the Oracle Solaris OS. The DTrace facility can be used to examine the behavior of your multithreaded program. DTrace inserts probes into running programs to collect data at points in the execution path that you specify. The collected data can be examined to determine problem areas. See the *Oracle Solaris 11.1 Dynamic Tracing Guide* for more information about using DTrace.

Profiling with Performance Analyzer

The Performance Analyzer tool, included in the Oracle Solaris Studio Studio software, can be used for extensive profiling of multithreaded and single threaded programs. The tool enables you to see in detail what a thread is doing at any given point. See the *Oracle Solaris Studio 12.3: Performance Analyzer* guide for more information.

Detecting Data Races and Deadlocks Using Thread Analyzer

The Oracle Solaris Studio software includes a tool called the Thread Analyzer. This tool enables you to analyze the execution of a multithreaded program. It can detect multithreaded programming errors such as data races or deadlocks in code that is written using the Pthread API, the Oracle Solaris thread API, OpenMP directives, Oracle parallel directives, Cray parallel directives, or a mix of these technologies.

See the *Oracle Solaris Studio 12.3: Debugging a Program With dbx* guide for more information.

Using `dbx`

The `dbx` utility is a debugger included in the Oracle Solaris Studio developer tools, available from `http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/index.html`. With the Oracle Solaris Studio `dbx` command-line debugger, you can debug and execute source programs that are written in C, C++, and Fortran. You can use `dbx` by starting it in a terminal window and interactively debugging your program with `dbx` commands. If you prefer a graphical interface, you can use the same `dbx` functionality in the Debugging windows of the Oracle Solaris Studio IDE (Integrated Development Environment). For a description of how to start `dbx`, see the `dbx`(1) man page. See the manual *Oracle Solaris Studio 12.3: Debugging a Program With dbx* for an overview of `dbx`. The Debugging features in the Oracle Solaris Studio IDE are described in the IDE online help.

See Chapter 11, "Debugging Multithreaded Applications," in *Oracle Solaris Studio 12.3: Debugging a Program With dbx* for detailed information about debugging multithreaded programs. The `dbx` debugger provides commands to manipulate event handlers for thread events, which are described in Appendix B, "Event Management," in *Oracle Solaris Studio 12.3: Debugging a Program With dbx*.

All the `dbx` options that are listed in Table 8–1 can support multithreaded applications.

TABLE 8–1 `dbx` Options for MT Programs

Option	Action
`cont at` *line* `[-sig` *signo id*`]`	Continues execution at *line* with signal *signo*. The *id*, if present, specifies which thread or LWP to continue. The default value is *all*.
`lwp [`*lwpid*`]`	Displays current LWP. Switches to given LWP [*lwpid*].
`lwps`	Lists all LWPs in the current process.
`next ...` *tid*	Steps the given thread. When a function call is skipped, all LWPs are implicitly resumed for the duration of that function call. Nonactive threads cannot be stepped.
`next ...` *lwpid*	Steps the given LWP. Does not implicitly resume all LWPs when skipping a function. The LWP on which the given thread is active. Does not implicitly resume all LWP when skipping a function.
`step...` *tid*	Steps the given thread. When a function call is skipped, all LWPs are implicitly resumed for the duration of that function call. Nonactive threads cannot be stepped.
`step...` *lwpid*	Steps the given LWP. Does not implicitly resume all LWPs when skipping a function.
`stepi...` *lwpid*	Steps machine instructions (stepping into calls) in the given LWP.
`stepi...` *tid*	Steps machine instructions in the LWP on which the given thread is active.
`thread [` *tid* `]`	Displays current thread, or switches to thread *tid*. In all the following variations, omitting the l *tid* implies the current thread.
`thread -info [` *tid* `]`	Prints everything known about the given thread.
`thread -blocks [` *tid* `]`	Prints all locks held by the given thread blocking other threads.
`thread -suspend [` *tid* `]`	Puts the given thread into suspended state, which prevents it from running. A suspended thread displays with an "S" in the `threads` listing.
`thread -resume [` *tid* `]`	Unsuspends the given thread so it resumes running.
`thread -hide [` *tid* `]`	*Hides* the given or current thread. The thread does not appear in the generic `threads` listing.

TABLE 8–1 `dbx` Options for MT Programs *(Continued)*

Option	Action
`thread -unhide [` *tid* `]`	*Unhides* the given or current thread.
`thread -unhide all`	*Unhides* all threads.
`threads`	Prints the list of all known threads.
`threads -all`	Prints threads that are not usually printed (zombies).
`threads -mode all|filter`	Controls whether `threads` prints all threads or filters threads by default. When filtering is on, threads that have been hidden by the `thread -hide` command are not listed.
`threads -mode auto|manual`	Enables automatic updating of the thread listing.
`threads -mode`	Echoes the current modes. Any of the previous forms can be followed by a thread or LWP ID to get the traceback for the specified entity.

Tracing and Debugging With the TNF Utilities

Although Dtrace, Performance Analyzer, Thread Analyzer, and `dbx` are more modern tools, you can also still use the older TNF utilities to trace, debug, and gather performance analysis information from your applications and libraries. The TNF utilities integrate trace information from the kernel as well as from multiple user processes and threads. The TNF utilities have long been included as part of the Solaris software. See the `tracing`(3TNF) man page for information about these utilities.

Using truss

See the `truss(1)`man page for information on tracing system calls, signals and user-level function calls.

Using mdb

For information about `mdb`, see the *Oracle Solaris Modular Debugger Guide*.

The following `mdb` commands can be used to access the LWPs of a multithreaded program.

`$l`	Prints the LWP ID of the representative thread if the target is a user process.
`$L`	Prints the LWP IDs of each LWP in the target if the target is a user process.
pid`::attach`	Attaches to process # *pid*.

`::release` Releases the previously attached process or core file. The process can subsequently be continued by `prun(1)` or it can be resumed by applying MDB or another debugger.

These commands to set conditional breakpoints are often useful.

[*addr*] `::bp` [+/-dDestT] [`-c` *cmd*] [`-n` *count*] *sym* `...`
Set a breakpoint at the specified locations.

addr `::delete` [*id* | `all`]
Delete the event specifiers with the given ID number.

CHAPTER 9

Programming Guidelines

This chapter provides some pointers on programming with threads. Most pointers apply to both Oracle Solaris and POSIX threads, but where utility differs, the behavior is noted. A change from single-threaded thinking to multithreaded thinking is emphasized in this chapter. The chapter discusses the following topics:

- "Rethinking Global Variables" on page 221
- "Providing for Static Local Variables" on page 222
- "Synchronizing Threads" on page 223
- "Avoiding Deadlock" on page 226
- "Some Basic Guidelines for Threaded Code" on page 228
- "Creating and Using Threads" on page 229
- "Working With Multiprocessors" on page 229
- "Examples of Threads Programs" on page 234

Rethinking Global Variables

Historically, most code has been designed for single-threaded programs. This code design is especially true for most of the library routines that are called from C programs. The following implicit assumptions were made for single-threaded code:

- When you write into a global variable and then, a moment later, read from the variable, what you read is exactly what you just wrote.
- A write into nonglobal, static storage, and moment later, a read from the variable results in a read of exactly what you just wrote.
- You do not need synchronization because concurrent access to the variable is not invoked.

The following examples discuss some of the problems that arise in multithreaded programs because of these assumptions, and how you can deal with the problems.

Traditional, single-threaded C and UNIX have a convention for handling errors detected in system calls. System calls can return anything as a functional value. For example, `write()`

returns the number of bytes that were transferred. However, the value -1 is reserved to indicate that something went wrong. So, when a system call returns -1, you know that the call failed.

EXAMPLE 9–1 Global Variables and *errno*

```
extern int errno;
...
if (write(file_desc, buffer, size) == -1) {
    /* the system call failed */
    fprintf(stderr, "something went wrong, "
        "error code = %d\n", errno);
    exit(1);
}
...
```

Rather than returning the actual error code, which could be confused with normal return values, the error code is placed into the global variable errno. When the system call fails, you can look in errno to find out what went wrong.

Now, consider what happens in a multithreaded environment when two threads fail at about the same time but with different errors. Both threads expect to find their error codes in errno, but one copy of errno cannot hold both values. This global variable approach does not work for multithreaded programs.

Threads solve this problem through a conceptually new storage class: thread-specific data. This storage is similar to global storage. Thread-specific data can be accessed from any procedure in which a thread might be running. However, thread-specific data is private to the thread. When two threads refer to the thread-specific data location of the same name, the threads are referring to two different areas of storage.

So, when using threads, each reference to errno is thread specific because each thread has a private copy of errno. A reference to errno as thread-specific is achieved in this implementation by making errno a macro that expands to a function call.

Providing for Static Local Variables

Example 9–2 shows a problem that is similar to the errno problem, but involving static storage instead of global storage. The function gethostbyname(3NSL) is called with the computer name as its argument. The return value is a pointer to a structure that contains the required information for contacting the computer through network communications.

EXAMPLE 9–2 The gethostbyname() Problem

```
struct hostent *gethostbyname(char *name) {
    static struct hostent result;
        /* Lookup name in hosts database */
        /* Put answer in result */
    return(&result);
```

EXAMPLE 9–2 The `gethostbyname()` Problem *(Continued)*

```
}
```

A pointer that is returned to a local variable is generally not a good idea. Using a pointer works in this example because the variable is static. However, when two threads call this variable at once with different computer names, the use of static storage conflicts.

Thread-specific data could be used as a replacement for static storage, as in the `errno` problem. But, this replacement involves dynamic allocation of storage and adds to the expense of the call.

A better way to handle this kind of problem is to make the caller of `gethostbyname()` supply the storage for the result of the call. An additional output argument to the routine enables the caller to supply the storage. The additional output argument requires a new interface to the `gethostbyname()` function.

This technique is used in threads to fix many of these problems. In most cases, the name of the new interface is the old name with "`_r`" appended, as in `gethostbyname_r`(3NSL).

Synchronizing Threads

The threads in an application must cooperate and synchronize when sharing the data and the resources of the process.

A problem arises when multiple threads call functions that manipulate an object. In a single-threaded world, synchronizing access to such objects is not a problem. However, as Example 9–3 illustrates, synchronization is a concern with multithreaded code. Note that the `printf(3S)` function is safe to call for a multithreaded program. This example illustrates what could happen if `printf()` were not safe.

EXAMPLE 9–3 `printf()` Problem

```
/* thread 1: */
    printf("go to statement reached");


/* thread 2: */
    printf("hello world");



printed on display:
    go to hello
```

Single-Threaded Strategy

One strategy is to have a single, application-wide mutex lock acquired whenever any thread in the application is running and released before it must block. Because only one thread can be accessing shared data at any one time, each thread has a consistent view of memory.

Because this strategy is effectively single-threaded, very little is gained by this strategy.

Reentrant Function

A better approach is to take advantage of the principles of modularity and data encapsulation. A reentrant function behaves correctly if called simultaneously by several threads. To write a reentrant function is a matter of understanding just what *behaves correctly* means for this particular function.

Functions that are callable by several threads must be made reentrant. To make a function reentrant might require changes to the function interface or to the implementation.

Functions that access global state, like memory or files, have reentrance problems. These functions need to protect their use of global state with the appropriate synchronization mechanisms provided by threads.

The two basic strategies for making functions in modules reentrant are code locking and data locking.

Code Locking

Code locking is done at the function call level and guarantees that a function executes entirely under the protection of a lock. The assumption is for all access to data to be done through functions. Functions that share data should execute under the same lock.

Some parallel programming languages provide a construct called a *monitor*. The monitor implicitly does code locking for functions that are defined within the scope of the monitor. A monitor can also be implemented by a mutex lock.

Functions under the protection of the same mutex lock or within the same monitor are guaranteed to execute atomically with respect to other functions in the monitor.

Data Locking

Data locking guarantees that access to a *collection* of data is maintained consistently. For data locking, the concept of locking code is still there, but code locking is around references to shared (global) data, only. For mutual exclusion locking, only one thread can be in the critical section for each collection of data.

Alternatively, in a multiple reader, single writer protocol, several readers can be allowed for each collection of data or one writer. Multiple threads can execute in a single module when the threads operate on different data collections. In particular, the threads do not conflict on a single collection for the multiple readers, single writer protocol. So, data locking typically allows more concurrency than does code locking.

What strategy should you use when using locks, whether implemented with mutexes, condition variables, or semaphores, in a program? Should you try to achieve maximum parallelism by locking only when necessary and unlocking as soon as possible, called *fine-grained locking*? Or should you hold locks for long periods to minimize the overhead of taking and releasing locks, called *coarse-grained locking*?

The granularity of the lock depends on the amount of data protected by the lock. A very coarse-grained lock might be a single lock to protect all data. Dividing how the data is protected by the appropriate number of locks is very important. Locking that is too fine-grained can degrade performance. The overhead associated with acquiring and releasing locks can become significant when your application contains too many locks.

The common wisdom is to start with a coarse-grained approach, identify bottlenecks, and add finer-grained locking where necessary to alleviate the bottlenecks. This approach is reasonably sound advice, but use your own judgment about finding the balance between maximizing parallelism and minimizing lock overhead.

Invariants and Locks

For both code locking and data locking, *invariants* are important to control lock complexity. An invariant is a condition or relation that is always true.

The definition is modified somewhat for concurrent execution: an invariant is a condition or relation that is true when the associated lock is being set. Once the lock is set, the invariant can be false. However, the code that holds the lock must reestablish the invariant before releasing the lock.

An invariant can also be a condition or relation that is true when a lock is being set. Condition variables can be thought of as having an invariant that is the condition.

EXAMPLE 9–4 Testing the Invariant With assert(3C)

```
mutex_lock(&lock);
while((condition)==FALSE)
    cond_wait(&cv,&lock);
assert((condition)==TRUE);
  .
  .
  .
mutex_unlock(&lock);
```

The `assert()` statement is testing the invariant. The `cond_wait()` function does not preserve the invariant, which is why the invariant must be reevaluated when the thread returns.

Another example is a module that manages a doubly linked list of elements. For each item on the list, a good invariant is the forward pointer of the previous item on the list. The forward pointer should also point to the same element as the backward pointer of the forward item.

Assume that this module uses code-based locking and therefore is protected by a single global mutex lock. When an item is deleted or added, the mutex lock is acquired, the correct manipulation of the pointers is made, and the mutex lock is released. Obviously, at some point in the manipulation of the pointers the invariant is false, but the invariant is reestablished before the mutex lock is released.

Avoiding Deadlock

Deadlock is a permanent blocking of a set of threads that are competing for a set of resources. Just because some thread can make progress does not mean that a deadlock has not occurred somewhere else.

The most common error that causes deadlock is *self deadlock* or *recursive deadlock*. In a self deadlock or recursive deadlock, a thread tries to acquire a lock already held by the thread. Recursive deadlock is very easy to program by mistake.

For example, assume that a code monitor has every module function grab the mutex lock for the duration of the call. Then, any call between the functions within the module protected by the mutex lock immediately deadlocks. If a function calls code outside the module that circuitously calls back into any method protected by the same mutex lock, the function deadlocks too.

The solution for this kind of deadlock is to avoid calling functions outside the module that might depend on this module through some path. In particular, avoid calling functions that call back into the module without reestablishing invariants and do not drop all module locks before making the call. Of course, after the call completes and the locks are reacquired, the state must be verified to be sure the intended operation is still valid.

An example of another kind of deadlock is when two threads, thread 1 and thread 2, acquire a mutex lock, A and B, respectively. Suppose that thread 1 tries to acquire mutex lock B and thread 2 tries to acquire mutex lock A. Thread 1 cannot proceed while blocked waiting for mutex lock B. Thread 2 cannot proceed while blocked waiting for mutex lock A. Nothing can change. So, this condition is a permanent blocking of the threads, and a deadlock.

This kind of deadlock is avoided by establishing an order in which locks are acquired, a *lock hierarchy*. When all threads always acquire locks in the specified order, this deadlock is avoided.

Adherence to a strict order of lock acquisition is not always optimal. For instance, thread 2 has many assumptions about the state of the module while holding mutex lock B. Giving up mutex lock B to acquire mutex lock A and then reacquiring mutex lock B in that order causes the thread to discard its assumptions. The state of the module must be reevaluated.

The blocking synchronization primitives usually have variants that attempt to get a lock and fail if the variants cannot get the lock. An example is `pthread_mutex_trylock()`. This behavior of primitive variants allows threads to violate the lock hierarchy when no contention occurs. When contention occurs, the held locks must usually be discarded and the locks reacquired in order.

Deadlocks Related to Scheduling

Because the order in which locks are acquired is not guaranteed, a problem can occur where a particular thread never acquires a lock.

This problem usually happens when the thread holding the lock releases the lock, lets a small amount of time pass, and then reacquires the lock. Because the lock was released, the appearance is that the other thread should acquire the lock. But, nothing blocks the thread holding the lock. Consequently, that thread continues to run from the time the thread releases the lock until the time the lock is reacquired. Thus, no other thread is run.

You can usually solve this type of problem by calling `sched_yield()`(3C) just before the call to reacquire the lock. The `sched_yield()` function allows other threads to run and to acquire the lock.

Because the time-slice requirements of applications are so variable, the system does not impose any requirements. Use calls to `sched_yield()` to make threads share time as you require.

Locking Guidelines

Follow these simple guidelines for locking.

- Try not to hold locks across long operations like I/O where performance can be adversely affected.
- Do not hold locks when calling a function that is outside the module and might reenter the module.
- In general, start with a coarse-grained approach, identify bottlenecks, and add finer-grained locking where necessary to alleviate the bottlenecks. Most locks are held for short amounts of time and contention is rare. So, fix only those locks that have measured contention.
- When using multiple locks, avoid deadlocks by making sure that all threads acquire the locks in the same order.

Finding Deadlocks

The Oracle Solaris Studio Thread Analyzer is a tool that you can use to find deadlocks in your program. The Thread Analyzer can detect potential deadlocks as well as actual deadlocks. A potential deadlock does not necessarily occur in a given run, but can occur in any execution of the program depending on the scheduling of threads and the timing of lock requests by the threads. An actual deadlock is one that occurs during the execution of a program, causing the threads involved to hang, but may or may not cause the whole process to hang.

See the *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide* for more information.

Some Basic Guidelines for Threaded Code

- Know what you are importing and know whether it is safe.

  A threaded program cannot arbitrarily enter nonthreaded code.
- Threaded code can safely refer to unsafe code only from the initial thread.

  References to unsafe code in this way ensures that the static storage associated with the initial thread is used only by that thread.
- When making a library safe for multithreaded use, do not thread global process operations.

  Do not change global operations, or actions with global side effects, to behave in a threaded manner. For example, if file I/O is changed to per-thread operation, threads cannot cooperate in accessing files.

  For thread-specific behavior, or *thread cognizant* behavior, use thread facilities. For example, when the termination of `main()` should terminate only the thread that is exiting `main()`.

  ```
  pthread_exit();
   /*NOTREACHED*/
  ```
- Oracle-supplied libraries are assumed to be *unsafe* unless explicitly documented as *safe*.

  If a reference manual entry does not explicitly state that an interface is *MT-Safe*, you should assume that the interface is *unsafe*.
- Use compilation flags to manage binary incompatible source changes. See Chapter 8, "Compiling and Debugging," for complete instructions.
  - `-mt` enables multithreading.
  - `-lpthreads` used with the `-mt` option links in the POSIX threads functions. This flag is needed only if your program uses pthreads functions.
  - When using `-mt`, the Oracle Solaris threads APIs will be linked automatically. Always use the `-mt` option instead of listing `-lthread` explicitly. The `libpthread` library provides an interface to `libthread`, so you still need `libthread` when using pthreads.

Creating and Using Threads

The threads packages cache the threads data structure and stacks so that the repetitive creation of threads can be reasonably inexpensive. However, creating and destroying threads as the threads are required is usually more expensive than managing a pool of threads that wait for independent work. A good example is an RPC server that creates a thread for each request and destroys the thread when the reply is delivered.

Thread creation has less overhead than the overhead of process creation. However, thread creation is not efficient when compared to the cost of creating a few instructions. Create threads for processing that lasts at least a couple of thousand machine instructions.

Working With Multiprocessors

Multithreading enables you to take advantage of multiprocessors, including multicore and multithreaded processors, primarily through parallelism and scalability. Programmers should be aware of the differences between the memory models of a multiprocessor and a uniprocessor.

Note – In this manual, whenever multiprocessors are discussed, the context applies also to multicore and multithreaded processors unless noted otherwise.

Memory consistency is directly interrelated to the processor that interrogates memory. For uniprocessors, memory is obviously consistent because only one processor is viewing memory.

To improve multiprocessor performance, memory consistency is relaxed. You cannot always assume that changes made to memory by one processor are immediately reflected in the other processors' views of that memory.

You can avoid this complexity by using synchronization variables when you use shared or global variables.

Memory barrier synchronization is sometimes an efficient way to control parallelism on multiprocessors.

Another multiprocessor issue is efficient synchronization when threads must wait until all threads have reached a common point in their execution.

Note – The issues discussed here are not important when the threads synchronization primitives are always used to access shared memory locations. See Chapter 4, "Programming with Synchronization Objects."

Underlying Architecture

Threads synchronize access to shared storage locations by using the threads synchronization routines. With threads synchronization, running a program on a shared-memory multiprocessor has the identical effect of running the program on a uniprocessor.

However, in many situations a programmer might be tempted to take advantage of the multiprocessor and use “tricks” to avoid the synchronization routines. As Example 9–5 and Example 9–6 show, such tricks can be dangerous.

An understanding of the memory models supported by common multiprocessor architectures helps to understand the dangers.

The major multiprocessor components are:

- The processors, including cores and their threads, which run the programs
- Store buffers, which connect the processors to their caches
- *Caches*, which hold the contents of recently accessed or modified storage locations
- Memory, which is the primary storage and is shared by all processors

In the simple traditional model, the multiprocessor behaves as if the processors are connected directly to memory: when one processor stores into a location and another processor immediately loads from the same location, the second processor loads what was stored by the first.

Caches can be used to speed the average memory access. The desired semantics can be achieved when the caches are kept consistent with the other caches.

A problem with this simple approach is that the processor must often be delayed to make certain that the desired semantics are achieved. Many modern multiprocessors use various techniques to prevent such delays, which unfortunately change the semantics of the memory model.

Two of these techniques and their effects are explained in the next two examples.

Shared-Memory Multiprocessors

Consider the purported solution to the producer and consumer problem that is shown in Example 9–5.

Although this program works on current SPARC-based multiprocessors, the program assumes that all multiprocessors have strongly ordered memory. This program is therefore not portable.

EXAMPLE 9–5 Producer and Consumer Problem: Shared Memory Multiprocessors

```
char buffer[BSIZE];
unsigned int in = 0;
unsigned int out = 0;

/* Thread 1 (producer) */                /* Thread 2 (consumer) */

void                                     char
producer(char item) {                    consumer(void) {
     do                                       char item;
     {                                        do
        ;/* nothing */                        {
     }                                           ;/* nothing */
     while                                    }
         ((in - out) == BSIZE);               while
                                                  ((in - out) == 0);
     buffer[in%BSIZE] = item;                 item = buffer[out%BSIZE];
     in++;                                    out++;
}                                        }
```

When this program has exactly one producer and exactly one consumer and is run on a shared-memory multiprocessor, the program appears to be correct. The difference between `in` and `out` is the number of items in the buffer.

The producer waits, by repeatedly computing this difference, until room is available in the buffer for a new item. The consumer waits until an item is present in the buffer.

Strongly-ordered memory makes a modification to memory on one processor immediately available to the other processors. For strongly ordered memory, the solution is correct even taking into account that `in` and `out` will eventually overflow. The overflow occurs as long as `BSIZE` is less than the largest integer that can be represented in a word.

Shared-memory multiprocessors do not necessarily have strongly ordered memory. A change to memory by one processor is not necessarily available immediately to the other processors. See what happens when two changes to different memory locations are made by one processor. The other processors do not necessarily detect the changes in the order in which the changes were made because memory modifications do not happen immediately.

First the changes are stored in *store buffers* that are not visible to the cache.

The processor checks these store buffers to ensure that a program has a consistent view. But, because store buffers are not visible to other processors, a write by one processor does not become visible until the processor writes to cache.

The synchronization primitives use special instructions that flush the store buffers to cache. See Chapter 4, "Programming with Synchronization Objects." So, using locks around your shared data ensures memory consistency.

When memory ordering is very relaxed, Example 9–5 has a problem. The consumer might see that `in` has been incremented by the producer before the consumer sees the change to the corresponding buffer slot.

This memory ordering is called *weak ordering* because stores made by one processor can appear to happen out of order by another processor. Memory, however, is always consistent from the same processor. To fix this inconsistency, the code should use mutexes to flush the cache.

Because the trend is toward relaxing memory order, programmers must be careful to use locks around all global or shared data.

As demonstrated by Example 9–5 and Example 9–6, locking is essential.

Peterson's Algorithm

The code in Example 9–6 is an implementation of Peterson's Algorithm, which handles mutual exclusion between two threads. This code tries to guarantee that only one thread is in the critical section. When a thread calls `mut_excl()`, the thread enters the critical section sometime "soon."

An assumption here is that a thread exits fairly quickly after entering the critical section.

EXAMPLE 9–6 Mutual Exclusion for Two Threads

```
void mut_excl(int me /* 0 or 1 */) {
    static int loser;
    static int interested[2] = {0, 0};
    int other; /* local variable */

    other = 1 - me;
    interested[me] = 1;
    loser = me;
    while (loser == me && interested[other])
        ;

    /* critical section */
    interested[me] = 0;
}
```

This algorithm works some of the time when the multiprocessor has strongly ordered memory.

Some multiprocessors, including some SPARC-based multiprocessors, have store buffers. When a thread issues a store instruction, the data is put into a store buffer. The buffer contents are eventually sent to the cache, but not necessarily right away. The caches on each of the processors maintain a consistent view of memory, but modified data does not reach the cache right away.

When multiple memory locations are stored into, the changes reach the cache and memory in the correct order, but possibly after a delay. SPARC-based multiprocessors with this property are said to have total store order (TSO).

Suppose you have a situation where one processor stores into location A and loads from location B. Another processor stores into location B and loads from location A. Either the first processor fetches the newly-modified value from location B, or the second processor fetches the newly-modified value from location A, or both. However, the case in which both processors load the old values cannot happen.

Moreover, with the delays caused by load and store buffers, the "impossible case" can happen.

What could happen with Peterson's algorithm is that two threads running on separate processors both enter the critical section. Each thread stores into its own slot of the particular array and then loads from the other slot. Both threads read the old values (0), each thread assumes that the other party is not present, and both enter the critical section. This kind of problem might not be detected when you test a program, but only occurs much later.

To avoid this problem use the threads synchronization primitives, whose implementations issue special instructions, to force the writing of the store buffers to the cache. See Chapter 4, "Programming with Synchronization Objects."

Parallelizing a Loop on a Shared-Memory Parallel Computer

In many applications, and especially numerical applications, while part of the algorithm can be parallelized, other parts are inherently sequential, as shown in the following table. The algorithm can use barrier synchronization to coordinate the parallel and sequential portions.

TABLE 9–1 Multithreaded Cooperation Through Barrier Synchronization

Sequential Execution	Parallel Execution
`Thread 1`	`Thread 2 through Thread n`
`while(many_iterations) {`	`while(many_iterations) {`
`    sequential_computation`	
`    --- Barrier ---`	`    --- Barrier ---`
`    parallel_computation`	`    parallel_computation`
`}`	`}`

For example, you might produce a set of matrixes with a strictly linear computation, and perform operations on the matrixes that use a parallel algorithm. You then use the results of these operations to produce another set of matrixes, operate on these matrixes in parallel, and so on.

The nature of the parallel algorithms for such a computation is that little synchronization is required during the computation. But, synchronization of all the threads is required to ensure that the sequential computation is finished before the parallel computation begins.

The barrier forces all the threads that are doing the parallel computation to wait until all involved threads have reached the barrier. When the threads have reached the barrier, the threads are released and begin computing together.

Examples of Threads Programs

This guide has covered a wide variety of important threads programming issues. Appendix A, “Extended Example: A Thread Pool Implementation,” provides a pthreads program example that uses many of the features and styles that have been discussed.

Further Reading

For more in-depth information about multithreading, see *Programming with Threads* by Steve Kleiman, Devang Shah, and Bart Smaalders (Prentice-Hall, published in 1995). Note that although the book is not current with changes to the Oracle Solaris OS, much of the conceptual information is still valid.

APPENDIX A

Extended Example: A Thread Pool Implementation

This appendix provides a sample implementation of a useful package of interfaces for multithreaded programming: a thread pool.

- "What is a Thread Pool?" on page 235
- "About the Thread Pool Example" on page 236
- "What the Thread Pool Example Shows" on page 246

What is a Thread Pool?

Threads provide a useful paradigm for an application to do many things at once: if you have something to do, create a thread to do it. Using threads can simplify the logic of the application and also take advantage of multiple processors, but creating too many threads can cause overall application performance problems due to contention for resources. The application may end up spending much of its time contending for resources, dealing with mutex locks for example, and less of its time actually doing useful work. Also, creating a thread, though cheaper than creating a process, is still an expense. Creating a thread to do a small amount of work is wasteful.

A thread pool manages a set of anonymous threads that perform work on request. The threads do not terminate right away. When one of the threads completes a task, the thread becomes idle, ready to be dispatched to another task. The overhead of creating and destroying threads is limited to creating and destroying just the number of active worker threads in the pool. The application can have more tasks than there are worker threads, and this is usually the case. Processor utilization and throughput are improved by reducing contention for resources. The submitted tasks are processed in order, usually faster than could be done by creating a thread per task.

About the Thread Pool Example

- "Thread Pool Functions" on page 236
- "Thread Pool Code Examples" on page 237

Thread Pool Functions

The `thr_pool.h` header file declares the following functional interfaces.

- "`thr_pool_create()`" on page 236
- "`thr_pool_queue()`" on page 236
- "`thr_pool_wait()`" on page 237
- "`thr_pool_destroy()`" on page 237

`thr_pool_create()`

Creates a thread pool. More than one pool can be created.

```
typedef struct thr_pool thr_pool_t;    /* opaque to clients */

thr_pool_t *thr_pool_create(uint_t min_threads, uint_t max_threads,
                uint_t linger, pthread_attr_t *attr);
```

min_threads	Minimum number of threads in the pool.
max_threads	Maximum number of threads in the pool.
linger	Number of seconds that idle threads can linger before exiting, when no tasks come in. The idle threads can only exit if they are extra threads, above the number of minimum threads.
attr	Attributes of all worker threads. This can be `NULL`.

On error, `thr_pool_create()` returns `NULL` with `errno` set to the error code.

`thr_pool_queue()`

Enqueue a work request or task to the thread pool job queue.

```
int  thr_pool_queue(thr_pool_t *pool, void *(*func)(void *), void *arg);
```

pool	A thread pool identifier returned from `thr_pool_create()`.
func	The task function to be called.
arg	The only argument passed to the task function.

On error, `thr_pool_queue()` returns -1 with `errno` set to the error code.

Notice the similarity of the *func* and *arg* arguments to the *start_routine* and *arg* arguments of `pthread_create()` shown in "`pthread_create` Syntax" on page 29. The `thr_pool_queue()` function can be used as a replacement for `pthread_create()` in existing applications. Note that if you use `thr_pool_queue()` instead of `pthread_create()`, you cannot use `pthread_join()` to wait for the task to complete.

`thr_pool_wait()`

Wait for all queued jobs to complete in the thread pool.

```
void  thr_pool_wait(thr_pool_t *pool);
```

pool is a thread pool identifier that is returned from `thr_pool_create()`.

`thr_pool_destroy()`

Cancel all queued jobs and destroy the pool. Worker threads that are actively processing tasks are cancelled.

```
extern    void    thr_pool_destroy(thr_pool_t *pool);
```

pool is a thread pool identifier that is returned from `thr_pool_create()`.

Thread Pool Code Examples

This section shows the code for the thread pool example:

- "`thr_pool.h` File" on page 237
- "`thr_pool.c` File" on page 238

`thr_pool.h` File

This file declares the functions used in the example.

EXAMPLE A–1 `thr_pool.h`

```
/*
 * Declarations for the clients of a thread pool.
 */

#include <pthread.h>

/*
 * The thr_pool_t type is opaque to the client.
 * It is created by thr_pool_create() and must be passed
 * unmodified to the remainder of the interfaces.
 */
typedef    struct thr_pool    thr_pool_t;
```

EXAMPLE A–1 `thr_pool.h` *(Continued)*

```
/*
 * Create a thread pool.
 *    min_threads:    the minimum number of threads kept in the pool,
 *            always available to perform work requests.
 *    max_threads:    the maximum number of threads that can be
 *            in the pool, performing work requests.
 *    linger:        the number of seconds excess idle worker threads
 *            (greater than min_threads) linger before exiting.
 *    attr:        attributes of all worker threads (can be NULL);
 *            can be destroyed after calling thr_pool_create().
 * On error, thr_pool_create() returns NULL with errno set to the error code.
 */
extern    thr_pool_t    *thr_pool_create(uint_t min_threads, uint_t max_threads,
                uint_t linger, pthread_attr_t *attr);

/*
 * Enqueue a work request to the thread pool job queue.
 * If there are idle worker threads, awaken one to perform the job.
 * Else if the maximum number of workers has not been reached,
 * create a new worker thread to perform the job.
 * Else just return after adding the job to the queue;
 * an existing worker thread will perform the job when
 * it finishes the job it is currently performing.
 *
 * The job is performed as if a new detached thread were created for it:
 *    pthread_create(NULL, attr, void *(*func)(void *), void *arg);
 *
 * On error, thr_pool_queue() returns -1 with errno set to the error code.
 */
extern    int    thr_pool_queue(thr_pool_t *pool,
            void *(*func)(void *), void *arg);

/*
 * Wait for all queued jobs to complete.
 */
extern    void    thr_pool_wait(thr_pool_t *pool);

/*
 * Cancel all queued jobs and destroy the pool.
 */
extern    void    thr_pool_destroy(thr_pool_t *pool);
```

`thr_pool.c` File

This file implements the thread pool.

EXAMPLE A–2 `thr_pool.c`

```
/*
 * Thread pool implementation.
 * See <thr_pool.h> for interface declarations.
 */
```

EXAMPLE A–2 `thr_pool.c` *(Continued)*

```
#if !defined(_REENTRANT)
#define    _REENTRANT
#endif

#include "thr_pool.h"
#include <stdlib.h>
#include <signal.h>
#include <errno.h>

/*
 * FIFO queued job
 */
typedef struct job job_t;
struct job {
    job_t    *job_next;        /* linked list of jobs */
    void    *(*job_func)(void *);    /* function to call */
    void    *job_arg;        /* its argument */
};

/*
 * List of active worker threads, linked through their stacks.
 */
typedef struct active active_t;
struct active {
    active_t    *active_next;    /* linked list of threads */
    pthread_t    active_tid;    /* active thread id */
};

/*
 * The thread pool, opaque to the clients.
 */
struct thr_pool {
    thr_pool_t    *pool_forw;    /* circular linked list */
    thr_pool_t    *pool_back;    /* of all thread pools */
    pthread_mutex_t    pool_mutex;    /* protects the pool data */
    pthread_cond_t    pool_busycv;    /* synchronization in pool_queue */
    pthread_cond_t    pool_workcv;    /* synchronization with workers */
    pthread_cond_t    pool_waitcv;    /* synchronization in pool_wait() */
    active_t    *pool_active;    /* list of threads performing work */
    job_t        *pool_head;    /* head of FIFO job queue */
    job_t        *pool_tail;    /* tail of FIFO job queue */
    pthread_attr_t    pool_attr;    /* attributes of the workers */
    int        pool_flags;    /* see below */
    uint_t        pool_linger;    /* seconds before idle workers exit */
    int        pool_minimum;    /* minimum number of worker threads */
    int        pool_maximum;    /* maximum number of worker threads */
    int        pool_nthreads;    /* current number of worker threads */
    int        pool_idle;    /* number of idle workers */
};

/* pool_flags */
#define    POOL_WAIT    0x01        /* waiting in thr_pool_wait() */
#define    POOL_DESTROY    0x02        /* pool is being destroyed */

/* the list of all created and not yet destroyed thread pools */
```

EXAMPLE A–2 `thr_pool.c` *(Continued)*

```
static thr_pool_t *thr_pools = NULL;

/* protects thr_pools */
static pthread_mutex_t thr_pool_lock = PTHREAD_MUTEX_INITIALIZER;

/* set of all signals */
static sigset_t fillset;

static void *worker_thread(void *);

static int
create_worker(thr_pool_t *pool)
{
    sigset_t oset;
    int error;

    (void) pthread_sigmask(SIG_SETMASK, &fillset, &oset);
    error = pthread_create(NULL, &pool->pool_attr, worker_thread, pool);
    (void) pthread_sigmask(SIG_SETMASK, &oset, NULL);
    return (error);
}

/*
 * Worker thread is terminating.  Possible reasons:
 * - excess idle thread is terminating because there is no work.
 * - thread was cancelled (pool is being destroyed).
 * - the job function called pthread_exit().
 * In the last case, create another worker thread
 * if necessary to keep the pool populated.
 */
static void
worker_cleanup(thr_pool_t *pool)
{
    --pool->pool_nthreads;
    if (pool->pool_flags & POOL_DESTROY) {
        if (pool->pool_nthreads == 0)
            (void) pthread_cond_broadcast(&pool->pool_busycv);
    } else if (pool->pool_head != NULL &&
        pool->pool_nthreads < pool->pool_maximum &&
        create_worker(pool) == 0) {
        pool->pool_nthreads++;
    }
    (void) pthread_mutex_unlock(&pool->pool_mutex);
}

static void
notify_waiters(thr_pool_t *pool)
{
    if (pool->pool_head == NULL && pool->pool_active == NULL) {
        pool->pool_flags &= ~POOL_WAIT;
        (void) pthread_cond_broadcast(&pool->pool_waitcv);
    }
}

/*
```

EXAMPLE A–2 `thr_pool.c` *(Continued)*

```
 * Called by a worker thread on return from a job.
 */
static void
job_cleanup(thr_pool_t *pool)
{
    pthread_t my_tid = pthread_self();
    active_t *activep;
    active_t **activepp;

    (void) pthread_mutex_lock(&pool->pool_mutex);
    for (activepp = &pool->pool_active;
        (activep = *activepp) != NULL;
        activepp = &activep->active_next) {
        if (activep->active_tid == my_tid) {
            *activepp = activep->active_next;
            break;
        }
    }
    if (pool->pool_flags & POOL_WAIT)
        notify_waiters(pool);
}

static void *
worker_thread(void *arg)
{
    thr_pool_t *pool = (thr_pool_t *)arg;
    int timedout;
    job_t *job;
    void *(*func)(void *);
    active_t active;
    timestruc_t ts;

    /*
     * This is the worker's main loop.  It will only be left
     * if a timeout occurs or if the pool is being destroyed.
     */
    (void) pthread_mutex_lock(&pool->pool_mutex);
    pthread_cleanup_push(worker_cleanup, pool);
    active.active_tid = pthread_self();
    for (;;) {
        /*
         * We don't know what this thread was doing during
         * its last job, so we reset its signal mask and
         * cancellation state back to the initial values.
         */
        (void) pthread_sigmask(SIG_SETMASK, &fillset, NULL);
        (void) pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
        (void) pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

        timedout = 0;
        pool->pool_idle++;
        if (pool->pool_flags & POOL_WAIT)
            notify_waiters(pool);
        while (pool->pool_head == NULL &&
            !(pool->pool_flags & POOL_DESTROY)) {
```

EXAMPLE A–2 `thr_pool.c` *(Continued)*

```
            if (pool->pool_nthreads <= pool->pool_minimum) {
                (void) pthread_cond_wait(&pool->pool_workcv,
                    &pool->pool_mutex);
            } else {
                (void) clock_gettime(CLOCK_REALTIME, &ts);
                ts.tv_sec += pool->pool_linger;
                if (pool->pool_linger == 0 ||
                    pthread_cond_timedwait(&pool->pool_workcv,
                    &pool->pool_mutex, &ts) == ETIMEDOUT) {
                    timedout = 1;
                    break;
                }
            }
        }
        pool->pool_idle--;
        if (pool->pool_flags & POOL_DESTROY)
            break;
        if ((job = pool->pool_head) != NULL) {
            timedout = 0;
            func = job->job_func;
            arg = job->job_arg;
            pool->pool_head = job->job_next;
            if (job == pool->pool_tail)
                pool->pool_tail = NULL;
            active.active_next = pool->pool_active;
            pool->pool_active = &active;
            (void) pthread_mutex_unlock(&pool->pool_mutex);
            pthread_cleanup_push(job_cleanup, pool);
            free(job);
            /*
             * Call the specified job function.
             */
            (void) func(arg);
            /*
             * If the job function calls pthread_exit(), the thread
             * calls job_cleanup(pool) and worker_cleanup(pool);
             * the integrity of the pool is thereby maintained.
             */
            pthread_cleanup_pop(1);    /* job_cleanup(pool) */
        }
        if (timedout && pool->pool_nthreads > pool->pool_minimum) {
            /*
             * We timed out and there is no work to be done
             * and the number of workers exceeds the minimum.
             * Exit now to reduce the size of the pool.
             */
            break;
        }
    }
    pthread_cleanup_pop(1);    /* worker_cleanup(pool) */
    return (NULL);
}

static void
clone_attributes(pthread_attr_t *new_attr, pthread_attr_t *old_attr)
```

EXAMPLE A–2 `thr_pool.c` *(Continued)*

```
{
    struct sched_param param;
    void *addr;
    size_t size;
    int value;

    (void) pthread_attr_init(new_attr);

    if (old_attr != NULL) {
        (void) pthread_attr_getstack(old_attr, &addr, &size);
        /* don't allow a non-NULL thread stack address */
        (void) pthread_attr_setstack(new_attr, NULL, size);

        (void) pthread_attr_getscope(old_attr, &value);
        (void) pthread_attr_setscope(new_attr, value);

        (void) pthread_attr_getinheritsched(old_attr, &value);
        (void) pthread_attr_setinheritsched(new_attr, value);

        (void) pthread_attr_getschedpolicy(old_attr, &value);
        (void) pthread_attr_setschedpolicy(new_attr, value);

        (void) pthread_attr_getschedparam(old_attr, &param);
        (void) pthread_attr_setschedparam(new_attr, &param);

        (void) pthread_attr_getguardsize(old_attr, &size);
        (void) pthread_attr_setguardsize(new_attr, size);
    }

    /* make all pool threads be detached threads */
    (void) pthread_attr_setdetachstate(new_attr, PTHREAD_CREATE_DETACHED);
}

thr_pool_t *
thr_pool_create(uint_t min_threads, uint_t max_threads, uint_t linger,
    pthread_attr_t *attr)
{
    thr_pool_t    *pool;

    (void) sigfillset(&fillset);

    if (min_threads > max_threads || max_threads < 1) {
        errno = EINVAL;
        return (NULL);
    }

    if ((pool = malloc(sizeof (*pool))) == NULL) {
        errno = ENOMEM;
        return (NULL);
    }
    (void) pthread_mutex_init(&pool->pool_mutex, NULL);
    (void) pthread_cond_init(&pool->pool_busycv, NULL);
    (void) pthread_cond_init(&pool->pool_workcv, NULL);
    (void) pthread_cond_init(&pool->pool_waitcv, NULL);
    pool->pool_active = NULL;
```

EXAMPLE A–2 `thr_pool.c` *(Continued)*

```
    pool->pool_head = NULL;
    pool->pool_tail = NULL;
    pool->pool_flags = 0;
    pool->pool_linger = linger;
    pool->pool_minimum = min_threads;
    pool->pool_maximum = max_threads;
    pool->pool_nthreads = 0;
    pool->pool_idle = 0;

    /*
     * We cannot just copy the attribute pointer.
     * We need to initialize a new pthread_attr_t structure using
     * the values from the caller-supplied attribute structure.
     * If the attribute pointer is NULL, we need to initialize
     * the new pthread_attr_t structure with default values.
     */
    clone_attributes(&pool->pool_attr, attr);

    /* insert into the global list of all thread pools */
    (void) pthread_mutex_lock(&thr_pool_lock);
    if (thr_pools == NULL) {
        pool->pool_forw = pool;
        pool->pool_back = pool;
        thr_pools = pool;
    } else {
        thr_pools->pool_back->pool_forw = pool;
        pool->pool_forw = thr_pools;
        pool->pool_back = thr_pools->pool_back;
        thr_pools->pool_back = pool;
    }
    (void) pthread_mutex_unlock(&thr_pool_lock);

    return (pool);
}

int
thr_pool_queue(thr_pool_t *pool, void *(*func)(void *), void *arg)
{
    job_t *job;

    if ((job = malloc(sizeof (*job))) == NULL) {
        errno = ENOMEM;
        return (-1);
    }
    job->job_next = NULL;
    job->job_func = func;
    job->job_arg = arg;

    (void) pthread_mutex_lock(&pool->pool_mutex);

    if (pool->pool_head == NULL)
        pool->pool_head = job;
    else
        pool->pool_tail->job_next = job;
    pool->pool_tail = job;
```

EXAMPLE A–2 `thr_pool.c` *(Continued)*

```
    if (pool->pool_idle > 0)
        (void) pthread_cond_signal(&pool->pool_workcv);
    else if (pool->pool_nthreads < pool->pool_maximum &&
        create_worker(pool) == 0)
        pool->pool_nthreads++;

    (void) pthread_mutex_unlock(&pool->pool_mutex);
    return (0);
}

void
thr_pool_wait(thr_pool_t *pool)
{
    (void) pthread_mutex_lock(&pool->pool_mutex);
    pthread_cleanup_push(pthread_mutex_unlock, &pool->pool_mutex);
    while (pool->pool_head != NULL || pool->pool_active != NULL) {
        pool->pool_flags |= POOL_WAIT;
        (void) pthread_cond_wait(&pool->pool_waitcv, &pool->pool_mutex);
    }
    pthread_cleanup_pop(1);    /* pthread_mutex_unlock(&pool->pool_mutex); */
}

void
thr_pool_destroy(thr_pool_t *pool)
{
    active_t *activep;
    job_t *job;

    (void) pthread_mutex_lock(&pool->pool_mutex);
    pthread_cleanup_push(pthread_mutex_unlock, &pool->pool_mutex);

    /* mark the pool as being destroyed; wakeup idle workers */
    pool->pool_flags |= POOL_DESTROY;
    (void) pthread_cond_broadcast(&pool->pool_workcv);

    /* cancel all active workers */
    for (activep = pool->pool_active;
        activep != NULL;
        activep = activep->active_next)
        (void) pthread_cancel(activep->active_tid);

    /* wait for all active workers to finish */
    while (pool->pool_active != NULL) {
        pool->pool_flags |= POOL_WAIT;
        (void) pthread_cond_wait(&pool->pool_waitcv, &pool->pool_mutex);
    }

    /* the last worker to terminate will wake us up */
    while (pool->pool_nthreads != 0)
        (void) pthread_cond_wait(&pool->pool_busycv, &pool->pool_mutex);

    pthread_cleanup_pop(1);    /* pthread_mutex_unlock(&pool->pool_mutex); */

    /*
```

EXAMPLE A–2 `thr_pool.c` *(Continued)*

```
     * Unlink the pool from the global list of all pools.
     */
    (void) pthread_mutex_lock(&thr_pool_lock);
    if (thr_pools == pool)
        thr_pools = pool->pool_forw;
    if (thr_pools == pool)
        thr_pools = NULL;
    else {
        pool->pool_back->pool_forw = pool->pool_forw;
        pool->pool_forw->pool_back = pool->pool_back;
    }
    (void) pthread_mutex_unlock(&thr_pool_lock);

    /*
     * There should be no pending jobs, but just in case...
     */
    for (job = pool->pool_head; job != NULL; job = pool->pool_head) {
        pool->pool_head = job->job_next;
        free(job);
    }
    (void) pthread_attr_destroy(&pool->pool_attr);
    free(pool);
}
```

What the Thread Pool Example Shows

The example illustrates cancellation and unexpected thread termination, which is one of the trickier aspects of programming with threads. A worker thread might exit by calling `pthread_exit()` from within the task function passed to `thr_pool_queue()`, rather than just returning from the task function as expected. The thread pool recovers from this by catching the termination in a `pthread_cleanup_push()` function. The only harm done is that another worker thread must then be created. Worker threads that are actively processing tasks are cancelled in `thr_pool_destroy()`. A caller of `thr_pool_wait()` or `thr_pool_destroy()` may be cancelled by the application while it is waiting. This is also dealt with by using `pthread_cleanup_push()`.

Although the example package is useful as it is, an application might require some features that are missing here, such as:

- `fork()` safety (with `pthread_atfork()`).
- Ability to wait for completion of individual tasks.
- Faster memory allocation (the sample code uses `malloc()`).

Index

Numbers and Symbols
32-bit architectures, 72
64-bit environment
  data type model, 24
  /dev/kmem, 24
  /dev/mem, 24
  large file support, 25
  large virtual address space, 24
  libkvm, 24
  libraries, 25
  /proc restrictions, 25
  registers, 25

A
accessing the signal mask, 43
Ada, 159
adding signals to mask, 44
aio_cancel(), 164
aio_error(), 163
aio_error() function, 163
aio_read() function, 163
aio_return(), 163
aio_return() function, 163
aio_suspend() function, 163
aio_waitn() function, 163
aio_write() function, 163
aiocbp, 163
alarms, 150
algorithms
  faster with MT, 20
algorithms *(Continued)*
  parallel, 234
  sequential, 234
allocating storage from heap, malloc, 31
application-level threads, 18
architecture
  multiprocessor, 230–234
  SPARC, 72, 230, 232
assert statement, 117, 118, 225
Async-Signal-Safe
  signal handlers, 159
  functions
    POSIX, 160
    Solaris, 208
  threads, 157
asynchronous
  event notification, 121
  I/O, 162
  semaphore use, 121
  signals, 154, 157
asynchronous I/O
  operations, 163
    waiting for, 163
atomic, defined, 72
automatic, stack allocation, 65

B
barrier synchronization, 137–141, 233
binary semaphores, 119

binding
  threads to LWPs, 181
  values to key, 187
bottlenecks, 227
bound threads, 18
  defined, 18

C
cache, defined, 230
caching, threads data structure, 229
cancellation of thread, 23
cancelling threads, 46
  cancellation points, 46
changing the signal mask, 43, 184
circularly-linked list, example, 97
coarse-grained locking, 225
code lock, 224, 225
code monitor, 224, 226
compare thread identifiers, 39
compiling a multithreaded application, 211
completion semantics, 158
cond_broadcast
  return values, 198
  syntax, 198
cond_destroy
  return values, 195
  syntax, 195
cond_init, 202
  return values, 195
  syntax, 194–195
  USYNC_THREAD, 202
cond_reltimedwait
  return values, 197
  syntax, 197
cond_signal
  return values, 198
  syntax, 198
cond_timedwait
  return values, 197
  syntax, 196–197
cond_wait
  return values, 196
  syntax, 196
condition variables, 72, 102–107, 118
  blocking for specified time, 113
  blocking on, 109
  blocking until specified time, 111
  destroying state, 115
  getting clock selection, 106
  getting scope, 105
  initializing, 108
  intializing attributes, 103
  removing attribute, 104
  setting clock selection, 106
  setting scope, 104
  unblocking one thread, 110
  unblocking threads, 114
condition wait, POSIX threads, 161
contention, 227
continuing execution, 173
counting semaphores, 18, 119
creating
  stacks, 66, 68, 180, 183
  threads, 229
creating a default thread, 28
critical section, 232
custom stack, 66, 182

D
daemon threads, 181
data
  lock, 224, 225
  races, 205
  shared, 23, 232
  thread-specific, 33
data race, 214
data races, 216
dbx, 182
dbx(1), 216–218
deadlock
  defined, 226
  detecting, 216
  recursive, 226
  scheduling, 227
debugging, 214
  asynchronous signals, 215

debugging *(Continued)*
dbx, 182
dbx(1), 216–218
deadlocks, 215
dtrace, 215
hidden gap in synchronization, 215
inadequate stack size, 215
large automatic arrays, 215
long-jumping without releasing mutex lock, 215
mdb(1), 218
no synchronization of global memory, 214
passing pointer to caller's stack, 214
recursive deadlock, 215
reevaluate conditions after return from condition wait, 215
deleting signals from mask, 44
destroying an existing thread-specific data key, 34
detached threads, 55, 181
Dijkstra, E. W., 119
dispatch priority, calculation, 152
dispatching priority, 151
DTrace, 215

E
errno, 37, 222
global variables, 221
event notification, 121
examining the signal mask, 43, 184
example, thread pool, 235
exec, 146, 148, 149
exit, 149, 181

F
fair share scheduler (FSS) scheduling class, 153
finding thread priority, 190
fine-grained locking, 225
fixed priority scheduling class (FX), 153
flags to thr_create, 181
flockfile, 165
fork, 148
fork1, 147, 148
funlockfile, 165

G
getc, 165
getc_unlocked, 165
gethostbyname, 222
gethostbyname_r, 223
getrusage, 151
getting thread-specific key binding, 36
global
data, 224
side effects, 228
state, 224
variables, 36, 221–222

I
I/O
asynchronous, 162
nonsequential, 164
standard, 164
synchronous, 162
inheriting priority, 180
initializing the mutex, 86
interrupt, 154
invariants, 118, 225

J
joining threads, 30, 45, 55, 185

K
key, binding value to key, 187
keys, storing value of, 188
kill(), 154, 156

L

libc, 209
libC, 209
libCrun, 209
libCstd, 209
libiostream, 209
libm, 209
libmalloc, 209
libmapmalloc, 209
libnsl, 209
libpthread, 19
libraries, MT-Safe, 209
library, C routines, 221
libresolv, 209
libsocket, 209
libthread, 19
libX11, 209
lightweight processes
  debugging, 218
  defined, 18
  scheduling classes, 151
limits, resources, 151
local variable, 223
lock hierarchy, 226
locking, 224
  coarse grained, 225, 227
  code, 224
  conditional, 95
  data, 224–225
  fine-grained, 225, 227
  guidelines, 227
  invariants, 225
locks, 72
  mutual exclusion, 72, 97
  read-write, 178
  readers/writer, 72
longjmp, 150, 159
lost wake-up, 115
-lposix4 library, POSIX 1003.1 semaphore, 214
lseek(2), 164
LWP, defined, 18

M

malloc, 31
MAP_NORESERVE, 65
MAP_SHARED, 148
mdb(1), 218
memory
  consistency, 229
  ordering, relaxed, 232
  strongly ordered, 231
mmap, 148
mmap(2), 65
monitor, 226
monitor, code, 224
mprotect, 183
MT-Safe libraries
  alternative mmap(2)-based memory allocation library, 209
  C ++ runtime shared objects
    for C++ 4.0 compiler, 209
  C++ runtime shared objects
    for C++ 5.0 compiler, 209
  C++ standard library
    for Oracle C++ 5.x compilers, 209
  classic iostreams
    for C++, 209
  math library, 209
  network interfaces of the form getXXbyYY_r, 209
  socket library for making network connections, 209
  space-efficient memory allocation, 209
  thread-safe form of unsafe interfaces, 209
  thread-specific errno support, 209
  X11 Windows routines, 209
multiple-readers, single-writer locks, 178
multiprocessors, 229–234
multithreading, defined, 18
mutex, defined, 18
mutex, mutual exclusion locks, 226
mutex_destroy
  return values, 192
  syntax, 192
mutex_init, 202
  return values, 191
  syntax, 190–191
  USYNC_THREAD, 202

mutex_lock
  return values, 193
  syntax, 192
mutex scope, 75
mutex_trylock
  return values, 194
  syntax, 193
mutex type
  PTHREAD_MUTEX_ERRORCHECK, 88
  PTHREAD_MUTEX_NORMAL, 88
  PTHREAD_MUTEX_RECUSIVE, 88
mutex_unlock, 193
  return values, 193
mutual exclusion locks, 72, 97
  attributes, 73
  deadlock, 94
  default attributes, 72
  destroying mutex, 75
  destroying mutex state, 93
  getting mutex robust attribute, 84
  getting mutex scope, 76
  getting priority ceiling of mutex, 82
  getting priority ceiling of mutex attribute, 81
  getting protocol of mutex attribute, 80
  initializing, 86
  locking, 88
  making consistent, 87
  nonblock locking, 90
  setting mutex robust attribute, 83
  setting priority ceiling of mutex, 82
  setting priority ceiling of mutex attribute, 80
  setting protocol of mutex attribute, 78
  setting type attribute, 77
  unlocking, 89

N
NDEBUG, 117
netdir, 209
netselect, 209
nice value and class priority, 152
nonsequential I/O, 164
null
  threads, 66, 182
null threads, 182

P
parallel, algorithms, 234
PC, program counter, 22
PC_GETCID, 151
PC_GETCLINFO, 151
PC_GETPARMS, 151
PC_SETPARMS, 151
Performance Analyzer, 216
Peterson's Algorithm, 232
PL/1 language, 155
portability, 72
pread, 164
printf, 160
  problem, 223
priocntl(), 152
priocntl, 151
  PC_GETCID, 151
  PC_GETCLINFO, 151
  PC_SETPARMS, 151
priocntl(2), PC_GETPARMS, 151
priority, 22
  and scheduling, 152, 189
  inheritance, 180, 190
  range, 189
  setting for a thread, 189
priority inversion, 79
producer and consumer problem, 125, 142, 230
producer/consumer problem, 202
profiling, 216
  multithreaded programs, 150
programmer-allocated stack, 66, 182, 183
prolagen, decrease semaphore, P operation, 119
pthread_atfork, 148
  syntax, 44, 148
pthread_attr_destroy
  return values, 54
  syntax, 54
pthread_attr_getdetachstate
  return values, 56
  syntax, 56

pthread_attr_getguardsize
- return values, 57
- syntax, 57

pthread_attr_getinheritsched
- return values, 63
- syntax, 62

pthread_attr_getschedparam
- return values, 65
- syntax, 64–65

pthread_attr_getschedpolicy
- return values, 61
- syntax, 61

pthread_attr_getscope
- return values, 59
- syntax, 58–59

pthread_attr_getstack
- return values, 69
- syntax, 69

pthread_attr_getstacksize
- return values, 68
- syntax, 67

pthread_attr_init
- attribute values, 53
- return values, 53
- syntax, 52–53

pthread_attr_setdetachstate
- return values, 55
- syntax, 54–55

pthread_attr_setguardsize
- return values, 57
- syntax, 56–57

pthread_attr_setinheritsched
- return values, 62
- syntax, 62

pthread_attr_setschedparam
- return values, 63
- syntax, 63

pthread_attr_setschedpolicy
- return values, 61
- syntax, 60–61

pthread_attr_setscope
- return values, 58
- syntax, 58

pthread_attr_setstack
- return values, 68
- syntax, 68

pthread_attr_setstacksize
- return values, 67
- syntax, 66–67

pthread_barrier_destroy
- return values, 139
- syntax, 139

pthread_barrier_init
- return values, 137–138
- syntax, 137

pthread_barrier_wait
- return values, 138
- syntax, 138

pthread_barrierattr_destroy
- return values, 141
- syntax, 141

pthread_barrierattr_getpshared, return values, 141

pthread_barrierattr_init
- return values, 140
- syntax, 139

pthread_barrierattr_setpshared, return values, 140

pthread_cancel
- return values, 47
- syntax, 47

pthread_cleanup_pop, syntax, 50

pthread_cleanup_push, syntax, 50

pthread_cond_broadcast, 109, 115, 154
- example, 114
- return values, 115
- syntax, 114–115

pthread_cond_destroy
- return values, 115
- syntax, 115

pthread_cond_init
- return values, 108
- syntax, 108

pthread_cond_reltimedwait_np
- return values, 113
- syntax, 113

pthread_cond_signal, 109, 115, 117, 154

pthread_cond_signal *(Continued)*
example, 111
return values, 111
syntax, 110–111
pthread_cond_timedwait
example, 112
return values, 112
syntax, 111–112
pthread_cond_wait, 116, 154
example, 111
return values, 110
syntax, 109–110
pthread_condattr_destroy
return values, 104
syntax, 104
pthread_condattr_getclock, return values, 107
pthread_condattr_getpshared
return values, 106
syntax, 105
pthread_condattr_init
return values, 104
syntax, 103
pthread_condattr_setclock
return values, 106
syntax, 106
pthread_condattr_setpshared
return values, 105
syntax, 104–105
pthread_create
return values, 29
syntax, 29
pthread_detach
return values, 32
syntax, 32
pthread_equal
return values, 39
syntax, 39
pthread_exit
return values, 45
syntax, 45
pthread_getconcurrency
return values, 60
syntax, 60
pthread_getschedparam
return values, 42
syntax, 41–42
pthread_getspecific, syntax, 36
pthread_join, 162
return values, 31
syntax, 30
pthread_join(3C), 65
pthread_key_create
example, 38
return values, 34
syntax, 33–34
pthread_key_delete
return values, 35
syntax, 34–35
pthread_kill(), 156
pthread_kill
return values, 43
syntax, 43
pthread_mutex_consistent_np
return values, 87
syntax, 87
pthread_mutex_destroy
return values, 93
syntax, 93
pthread_mutex_getprioceiling
return values, 83
syntax, 82
pthread_mutex_init
return values, 86
syntax, 86
pthread_mutex_lock
example, 93, 97
return values, 88
syntax, 88
pthread_mutex_setprioceiling
return values, 82
syntax, 82
pthread_mutex_timedlock
return values, 92
syntax, 91, 92
pthread_mutex_trylock, 96
return values, 90
syntax, 90

pthread_mutex_trylock(3C), 227
pthread_mutex_unlock
  example, 93, 97
  return values, 90
  syntax, 89–90
pthread_mutexattr_destroy
  return values, 75
  syntax, 75
pthread_mutexattr_getprioceiling
  return values, 81
  syntax, 81
pthread_mutexattr_getprotocol
  return values, 80
  syntax, 80
pthread_mutexattr_getpshared
  return values, 76
  syntax, 76
pthread_mutexattr_getrobust_np
  return values, 85
  syntax, 84
pthread_mutexattr_gettype
  return values, 78
  syntax, 77
pthread_mutexattr_init
  return values, 74
  syntax, 73–74
pthread_mutexattr_setprioceiling
  return values, 81
  syntax, 80
pthread_mutexattr_setprotocol
  return values, 79
  syntax, 78
pthread_mutexattr_setpshared
  return values, 76
  syntax, 75
pthread_mutexattr_setrobust_np
  return value, 84
  syntax, 83
pthread_mutexattr_settype
  return values, 77
  syntax, 76
pthread_once
  return values, 40
  syntax, 39–40
PTHREAD_PRIO_INHERIT, 79
PTHREAD_PRIO_NONE, 78
PTHREAD_PRIO_PROTECT, 79
pthread_rwlock_destroy
  return values, 136
  syntax, 136
pthread_rwlock_init
  return values, 130
  syntax, 130
pthread_rwlock_rdlock
  return values, 131
  syntax, 130–131
pthread_rwlock_timedrdlock
  return values, 132
  syntax, 131–132
pthread_rwlock_timedwrlock, syntax, 134
pthread_rwlock_tryrdlock
  return values, 133
  syntax, 132
pthread_rwlock_trywrlock
  return values, 134
  syntax, 134
pthread_rwlock_unlock
  return values, 136
  syntax, 135–136
pthread_rwlock_wrlock
  return values, 133
  syntax, 133
pthread_rwlockattr_destroy
  return values, 128
  syntax, 127
pthread_rwlockattr_getpshared
  return values, 129
  syntax, 129
pthread_rwlockattr_init
  return values, 127
  syntax, 127
pthread_rwlockattr_setpshared
  return values, 128
  syntax, 128
PTHREAD_SCOPE_PROCESS, 23
PTHREAD_SCOPE_SYSTEM, 23, 57
pthread_self
  return values, 39

pthread_self *(Continued)*
syntax, 38–39
pthread_setcancelstate
return values, 48
syntax, 48
pthread_setcanceltype
return values, 49
syntax, 48–49
pthread_setconcurrency
return values, 59
syntax, 59
pthread_setschedparam
return values, 41
syntax, 40–41
pthread_setschedprio
return values, 42
syntax, 42
pthread_setspecific
example, 38
return values, 35
syntax, 35
pthread_sigmask(), 156
pthread_sigmask
return values, 44
syntax, 43–44
pthread_spin_destroy()
syntax, 101–102, 102
pthread_spin_init, syntax, 99
pthread_spin_lock
return values, 100
syntax, 100
pthread_spin_trylock
return values, 101
syntax, 100
pthread_spin_unlock
return values, 101
syntax, 101
PTHREAD_STACK_MIN, 66
pthread_testcancel, syntax, 49
putc, 165
putc_unlocked, 165
pwrite, 164

R
_r, 223
read, 164
read-write locks, 178
acquiring read lock, 130
acquiring write lock, 133
attributes, 126
destroying, 136
destroying lock attribute, 127
getting lock attribute, 128
initializing lock, 127, 129
locking read lock, 132
locking write lock, 134
releasing read lock, 135
setting lock attribute, 128
readers/writer locks, 72
realtime, scheduling, 152
red zone, 65, 183
reentrant, 224
described, 224–226
functions, 207
strategies for making, 224
register state, 22
relaxed memory ordering, 232
remote procedure call RPC, 21
replacing signal mask, 44
resuming execution, 173
RPC, 21, 209, 229
RT,, *See* realtime
rw_rdlock
return values, 175
syntax, 175
rw_tryrdlock
return values, 176
syntax, 176
rw_trywrlock
return values, 177
syntax, 177
rw_unlock
return values, 178
syntax, 178
rw_wrlock
return values, 177
syntax, 176–177

rwlock_destroy
  return values, 179
  syntax, 178–179
rwlock_init
  return values, 175
  syntax, 174–175
  USYNC_THREAD, 202

S

SA_RESTART, 161
safety, threads interfaces, 205–209, 209
sched_yield, 227
  return values, 40
  syntax, 40
scheduling
  classes, 23
  policies, 23
  scopes, 23
  system class, 151
scheduling class
  fair share scheduler (FSS), 153
  fixed priority scheduler (FX), 153
  priority, 151
  realtime, 152
  timeshare, 152
scope, state, 23
sem_destroy
  return values, 125
  syntax, 124–125
sem_init
  example, 125
  return values, 122
sem_post, 119
  example, 125
  return values, 123
  syntax, 123
sem_trywait, 119
  return values, 124
  syntax, 124
sem_wait, 119
  example, 125
  return values, 123
  syntax, 123
sema_destroy
  return values, 202
  syntax, 202
sema_init
  return values, 200
  syntax, 121–122, 199–200
  USYNC_THREAD, 202
sema_post, 208
  return values, 200
  syntax, 200
sema_trywait
  return values, 201
  syntax, 201
sema_wait
  return values, 201
  syntax, 201
semaphores, 72, 119–126, 143
  binary, 119
  blocking calling thread, 123
  counting, 119
  counting, defined, 18
  decrement semaphore value, 119
  decrementing count, 124
  destroying state, 124
  increment semaphore value, 119
  incrementing, 122
  initializing, 121
  interprocess, 122
  intraprocess, 122
  named, 120
  unnamed, 120
sending signal to thread, 43
sequential algorithms, 233
setjmp, 150, 158, 159
setting thread-specific key binding, 35
shared data, 23, 224
shared-memory multiprocessor, 231
SIG_DFL, 153
SIG_IGN, 153
SIG_SETMASK, 44
sigaction(), 153
SIGFPE, 154, 159
SIGILL, 154
SIGINT, 154, 158

SIGIO, 154
siglongjmp, 159
signal(), 153
signal
  handler, 153, 157
signal.h, 184
signals
  access mask, 184
  adding to mask, 44
  asynchronous, 154, 157
  deleting from mask, 44
  inheritance, 180
  masks, 22
  pending, 173, 180
  replacing current mask, 44
  sending to a thread, 156
  sending to thread, 43, 184
  SIG_BLOCK, 44
  SIG_SETMASK, 44
  SIG_UNBLOCK, 44
  SIGSEGV, 65
  synchronous, 154
  unmasked and caught, 161
sigprocmask(), 156
sigqueue(), 154
SIGSEGV, 65, 154
sigsend(), 154
sigsetjmp, 159
sigtimedwait(), 157
sigwait(), 156–157, 157
sigwait, 159
single-threaded
  assumptions, 221
  code, 72
  defined, 18
  processes, 149
singly-linked list, example, 97
singly-linked list with nested locking, example, 97
size of stack, 66, 180, 182
spin locks, defined, 98–102
spurious wakeup, 161
stack, 229
  address, 68, 180
  boundaries, 65
stack *(Continued)*
  creation, 68, 180
  custom, 182
  deallocation, 183
  minimum size, 66
  overflows, 65
  pointer, 22
  programmer-allocated, 66, 182, 183
  red zone, 65, 183
  returning a pointer to, 207
  size, 66, 180
stack_base, 68, 180
stack_size, 66, 180
stack size
  finding minimum, 182
  minimum, 182
standard I/O, 164
standards, UNIX, 19
start_routine(), 181
static storage, 221
store buffer, 232
storing thread key value, 188
streaming a tape drive, 162
strongly ordered memory, 231
suspending a new thread, 181
swap space, 65
synchronization objects
  condition variables, 72, 102–107, 118
  mutex locks, 72, 97
  read-write locks, 178
  semaphores, 72, 119–126, 199, 203
synchronous I/O, 162
synchronous I/O, 162
synchronous signals, 154
system calls, handling errors, 221
system scheduling class, 151

T
tape drive, streaming, 162
THR_BOUND, 181
thr_continue, 181
  return values, 173
  syntax, 173

thr_create, 182
  return values, 182
  syntax, 180–182
THR_DAEMON, 181
THR_DETACHED, 181
thr_exit, 181
  return values, 185
  syntax, 185
thr_getprio
  return values, 190
  syntax, 190
thr_getspecific
  return values, 188
  syntax, 188
thr_join
  return values, 186
  syntax, 185
thr_keycreate
  return values, 187
  syntax, 187
thr_kill, 208
  return values, 184
  syntax, 184
thr_min_stack, 180, 182
thr_self, syntax, 183
thr_setprio
  return values, 189
  syntax, 189
thr_setspecific
  return values, 188
  syntax, 188
thr_sigsetmask, 208
  return values, 185
  syntax, 184–185
thr_suspend
  return values, 172
  syntax, 172
thr_yield, syntax, 183
Thread Analyzer, 216
thread create, exit status, 29
thread-directed signal, 157
thread identifier, 38–39
thread lifecycle, 27
thread local storage, 33
thread pool, defined, 235
thread-private storage, 22
thread-specific data, 33
  example, 36–38
  getting, 188
  global into private, 37
  new storage class, 222
  setting, 187
thread-specific keys
  creating, 33–34, 187
thread synchronization
  condition variables, 24
  mutex locks, 24
  mutual exclusion locks, 72
  read-write locks, 126
  read/write locks, 24
  semaphores, 24, 119–126
threads
  acquiring identifiers, 183
  cancelling, 46, 47
  creating, 180, 182, 229
  daemon, 181
  detached, 55, 181
  identifiers, 181
  joining, 185
  key, 187
  null, 66, 182
  priority, 180
  safety, 205, 209
  sending signal to, 184
  signals, 161
  stack, 207
  suspended, 173
  suspending, 181
  synchronization models, 24
  synchronizing, 72, 143
  terminating, 45, 185
  thread-specific data, 222
  user-level, 18, 22
  yielding execution, 183
threads defined, 17
time-out, example, 113
timer, per LWP, 149
timeshare scheduling class, 152

TLI, 209
tools
  dbx, 182
  dbx(1), 216–218
  Dtrace, 215
  mdb(1), 218
  Performance Analyzer, 216
  Thread Analyzer, 216
total store order, 233
trap, 154
  default action, 154
  handled by thread, 154
TS, *See* timeshare scheduling class

U
unbound threads
  caching, 229
  defined, 18
user-level threads, 18, 22
/usr/include/errno.h, 212
/usr/include/limits.h, 212
/usr/include/pthread.h, 211
/usr/include/signal.h, 212
/usr/include/thread.h, 211
/usr/include/unistd.h, 212
/usr/lib, 32–bit threads library, Solaris 9, 214
/usr/lib/lwp, 32-bit threads library, Solaris 8, 214
/usr/lib/lwp/64, 64-bit threads library, Solaris 8, 214
USYNC_PROCESS, 202
  condition variable, 194
  mutex, 190
  read-write lock, 174
  semaphore, 199
USYNC_PROCESS_ROBUST, mutex, 190
USYNC_THREAD
  condition variable, 194
  mutex, 190
  read-write lock, 174
  semaphore, 199

V
variables
  condition, 72, 102–107, 118, 143
  global, 221–222
  primitive, 72
verhogen, increase semaphore, V operation, 119
vfork, 147

W
write(2), 164

X
XDR, 209