IEEE Std 1003.1d-1999
(Amendment to
IEEE Std 1003.1-1990)

IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API)—Amendment d: Additional Realtime Extensions [C Language]

Sponsor

Portable Application Standards Committee of the IEEE Computer Society

Approved 16 September 1999

IEEE-SA Standards Board

Abstract: This standard is part of the POSIX series of standards for applications and user interfaces to open systems. It defines the applications interface to system services for spawning a process, timeouts for blocking services, sporadic server scheduling, execution time clocks and timers, and advisory information for file management. This standard is stated in terms of its C binding.

Keywords: API, application portability, C (programming language), data processing, open systems, operating system, portable application, POSIX, realtime

---

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Print: ISBN 0-7381-1815-X SH94790
PDF: ISBN 0-7381-1816-8 SS94790

Secretary, IEEE-SA Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

Contents

FIGURES

TABLES

Introduction

(This introduction is not a normative part of IEEE Std 1003.1d-1999, Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API)—Amendment d: Additional Realtime Extensions [C Language])

Editor's Note: This introduction consists of material that will eventually be integrated into the base POSIX.1[1] *standard's introduction (and the portion of Annex B that contains general rationale about the standard). The introduction contains text that was previously held in either the foreword or scope. As this portion of the standard is for information only, specific details of the integration with POSIX.1 are left as an editorial exercise. The section and subclause structure of this document follows that of POSIX.1. Sections that are not amended by this standard are omitted.*

The purpose of this document is to supplement the base standard with interfaces and functionality for applications having realtime requirements.

This standard defines systems interfaces to support the source portability of applications with realtime requirements. The system interfaces are all extensions of or additions to ISO/IEC 9945-1: 1990, *Portable Operating System Interface for Computer Environments*, as amended by POSIX.1b and POSIX.1c. Although rooted in the culture defined by ISO/IEC 9945-1: 1990, the interfaces are focused upon the realtime application requirements, which were beyond the ISO/IEC 9945-1: 1990 scope. The interfaces included in this standard are additions to the set required to make ISO/IEC 9945-1: 1990 minimally usable to realtime applications on single processor systems.

The definition of *realtime* used in defining the scope of this standard is

> Realtime in operating systems: the ability of the operating system to provide a required level of service in a bounded response time.

The key elements of defining the scope are

(1) defining a sufficient set of functionality to cover the realtime application program domain in the areas not covered by POSIX.1b and POSIX.1c;

(2) defining sufficient performance constraints and performance-related functions to allow a realtime application to achieve deterministic response from the system; and

(3) specifying changes or additions to improve or complete the definition of the facilities specified in the previous real-time or threads extensions covered by POSIX.1b and POSIX.1c.

Wherever possible, the requirements of other application environments were included in the interface definition. The specific areas are noted in the scope overviews of each of the interface areas given below.

The specific functional areas included in this standard and their scope include

— Spawn a process: new system services to spawn the execution of a new process in an efficient manner.

— Timeouts for some blocking services: additional services that provide a timeout capability to system services already defined in POSIX.1b and POSIX.1c, thus

---

1) See 2.3.3 in this standard for more information about these references.

allowing the application to include better error detection and recovery capabilities.

— Sporadic server scheduling: the addition of a new scheduling policy appropriate for scheduling aperiodic processes or threads in hard realtime applications.

— Execution time clocks and timers: the addition of new clocks that measure the execution times of processes or threads, and the possibility to create timers based upon these clocks, for runtime detection (and treatment) of execution time overruns.

— Advisory information for file management: addition of services that allow the application to specify advisory information that can be used by the system to achieve better or even deterministic response times in file management or input and output (I/O) operations.

There are two other functional areas that were included in the scope of this standard, but the balloting group considered that they were not ready yet for standardization:

— Device control: a new service to pass control information and commands between the application and device drivers.

— Interrupt control: new services that allow the application to directly handle hardware interrupts.

This standard has been defined exclusively at the source code level for the C programming language. Although the interfaces will be portable, some of the parameters used by an implementation may have hardware or configuration dependencies.

Related Standards Activities

Activities to extend this standard to address additional requirements are in progress, and similar efforts can be anticipated in the future.

The following areas are under active consideration at this time or are expected to become active in the near future:[2)]

(1) Additional system application program interfaces (APIs) in C language

(2) Ada and FORTRAN language bindings to (1)

(3) Shell and utility facilities

(4) Verification testing methods

(5) Realtime facilities

(6) Tracing facilities

(7) Fault tolerance

(8) Checkpoint/restart facilities

(9) Resource limiting facilities

(10) Network interface facilities

(11) System administration

---

2) A *Standards Status Report* that lists all current IEEE Computer Society standards projects is available from the IEEE Computer Society, 1730 Massachusetts Avenue NW, Washington, DC 20036-1903; Telephone: +1 202 371-0101; FAX: +1 202 728-9614. Working drafts of POSIX standards under development are available from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ 08855-1331 (http://www.standards.ieee.org/).

(12) Profiles describing application- or user-specific combinations of Open Systems standards

(13) An overall guide to POSIX-based or -related Open Systems standards and profiles

Extensions are approved as "amendments" or "revisions" to this document, following the IEEE and ISO/IEC procedures.

Approved amendments are published separately until the full document is reprinted and such amendments are incorporated in their proper positions.

If you have interest in participating in the Portable Application Standards Committee (PASC) working groups addressing these issues, please send your name, address, and phone number to

Secretary, IEEE Standards Board
Institute of Electrical and Electronics Engineers, Inc.
P.O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331
USA

When writing, ask to have your letter forwarded to the chairperson of the appropriate PASC working group. If you have interest in participating in this work at the international level, contact your International Organization for Standardization/International Electrotechnical Committee (ISO/IEC) national body.

This standard was prepared by the system services working group—realtime, sponsored by the Portable Application Standards Committee of the IEEE Computer Society. At the time this standard was approved, the membership of the system services working group—realtime was as follows:

Portable Application Standards Committee

Chair:	Lowell Johnson
Vice Chair:	Joseph M. Gwinn
Functional Vice Chairs:	Jay Ashford
	Andrew Josey
	Curtis Royster Jr.
Secretary:	Nick Stoughton

IEEE System Services Working Group—Realtime

Chair:	Joseph M. Gwinn
	Susan Corwin (to 1995)
Secretary:	Karen D. Gordon
	Franklin C. Prindle (1996)
	Lee Schemerhorn (to 1995)
Editor:	Michael González
	Robert D. Luken (to 1997)
Technical reviewers:	Steve Brosky
	Peter Dibble
	Christoph Eck
	Michael González
	Karen D. Gordon
	Joseph M. Gwinn
	Franklin C. Prindle
Ballot coordinators:	James T. Oblinger
	Duane Hughes (to 1996)

Working Group

Ray Alderman	Bill Gallmeister	Kent Long
Larry Anderson	Michael González	Robert D. Luken
Pierre-Jean Arcos	Karen D. Gordon	James T. Oblinger
Charles R. Arnold	Randy Greene	Offer Pazy
V. Raj Avula	Rick Greer	Franklin C. Prindle
Theodore P. Baker	Joseph M. Gwinn	François Riche
Robert Barned	Steven A. Haaser	Gordon W. Ross
Richard M. Bergman	Barbara Haleen	Curtis Royster, Jr.
Nawaf Bitar	Geoffrey R. Hall	Webb Scales
Steve Brosky	Patrick Hebert	Lee Schermerhorn
David Butenhof	Mary R. Hermann	Lui Sha
Hans Petter Christiansen	David Hughes	Del Swanson
Susan Corwin	Duane Hughes	Barry Traylor
Bill Cox	Michael B. Jones	Stephen R. Wali
Peter Dibble	Steve Kleiman	Andrew E. Wheeler, Jr.
Christoph Eck	Robert Knighten	David Wilner
Michael Feustel	C. Douglass Locke	John Zolnowsky

The following members of the balloting committee voted on this standard:

Phillip R. Acuff
Alejandro Alonso-Muñoz
Pierre-Jean Arcos
Jay Ashford
Theodore P. Baker
Robert Barned
Barbara K. Beauchamp
Keith Bierman
Nawaf Bitar
David Black
David J. Blackwood
Shirley Bockstahler-Brandt
Mark Brown
Alan Burns
Gregory Bussiere
H. L. Catala
Andrew B. Cheese
Michael W. Condry
Donald Cragun
John S. Davies
Richard P. Draves
Christoph Eck
Philip H. Enslow
W. Douglas Findley, Jr.
Bill Gallmeister
Michel P. Gien

Michael González
Karen D. Gordon
Mars J. Gralia
Joseph M. Gwinn
Steven A. Haaser
Chris J. Harding
Barry Hedquist
Karl Heubaum
Andrew R. Huber
Duane Hughes
Petr Janecek
Lowell G. Johnson
Michael B. Jones
Andrew Josey
Michael J. Karels
James J. Keys
Martin J. Kirk
Thomas M. Kurihara
Mark Larsen
Martin Leisner
Bruce Lewis
C. Douglass Locke
Roger J. Martin
Finnbarr P. Murphy
Richard E. Neese
James T. Oblinger

Diane Paul
Charles Pfleeger
John Pijanowski
Franklin C. Prindle
Juan Antonio de la Puente
François Riche
Chuck Roark
Hyman Rosen
Helmut Roth
Curtis Royster
Richard Scalzo
Richard Seibel
Keith Shillington
W. Olin Sibert
Jacob Slonim
Nicholas M. Stoughton
Gregory Swain
Efstathios D. Sykas
Donn S. Terry
Mark-Rene Uchida
Michael W. Vannier
Charlotte Wales
Frederick N. Webb
Laurence Wolfe
Oren Yuen
Ming De Zhou

The following organizational representatives voted on this standard:

James T. Oblinger
NGCR OSSWG

Diane Paul
SAE

Andrew Josey
X/Open Co. Ltd.

When the IEEE-SA Standards Board approved this standard on 16 September 1999, it had the following membership:

Richard J. Holleman, *Chair*
Donald N. Heirman, *Vice Chair*
Judith Gorman, *Secretary*

Satish K. Aggarwal
Dennis Bodson
Mark D. Bowman
James T. Carlo
Gary R. Engmann
Harold E. Epstein
Jay Forster*
Ruben D. Garzon

James H. Gurney
Lowell G. Johnson
Robert Kennelly
E.G. “Al” Kiener
Joseph L. Koepfinger*
L. Bruce McClung
Daleep C. Mohla
Robert F. Munzner

Louis-François Pau
Ronald C. Petersen
Gerald H. Peterson
John B. Posey
Gary S. Robinson
Akio Tojo
Hans E. Weinrich
Donald W. Zispe

*Member emeritus

Also included is the following nonvoting IEEE-SA Standards Board liaison:

Robert E. Hebner

Yvette Ho Sang
IEEE Standards Project Editor

IEEE Standard for Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API)—Amendment d: Additional Realtime Extensions [C Language]

Section 1: General

1.1 Scope

This standard defines realtime extensions to a standard operating system interface and environment to support application portability at the source-code level. It is intended to be used by both application developers and system implementers.

This standard will not change the base standard that it amends (including any existing amendments) in such a way as to cause implementations or strictly conforming applications to no longer conform.

The scope is to take existing realtime operating system practice and add it to the base standard. The definition of *realtime* used in defining the scope of this standard is

> **Realtime in operating systems:** the ability of the operating system to provide a required level of service in a bounded response time.

The key elements of defining the scope are

(1) defining a sufficient set of functionality to cover a significant part of the realtime application programming domain, and

(2) defining sufficient performance constraints and performance related functions to allow a realtime application to achieve deterministic response from the system.

Specifically within the scope is to define interfaces that do not preclude high performance implementations on traditional uniprocessor realtime systems. Wherever possible, the requirements of other application environments were included in the interface definition. The specific functional areas included in this document and their scope include

— Spawn: A process creation primitive useful for systems that have difficulty with *fork*() and as an efficient replacement for *fork*()/ *exec*.

— Timeouts: Alternatives to blocking primitives that provide a timeout parameter to be specified.

— Execution time monitoring: A set of execution time monitoring primitives that allow on-line measuring of thread and process execution times.

— Sporadic server: A scheduling policy for threads and processes that reserves a certain amount of execution capacity for processing aperiodic events at a given priority level.

— Advisory information: An interface that advises the implementation on (portable) application behavior so that it can optimize the system.

Two other functional areas were included in the scope of this standard, but the balloting group considered that they were not ready yet for standardization

— Device control: A portable application interface to nonportable special devices.

— Interrupt control: An interface that allows a process or thread to capture an interrupt, to block awaiting the arrival of an interrupt, and to protect critical sections of code that are contended for by a user-written interrupt service routine.

This standard has been defined exclusively at the source code level. Additionally, although the interfaces will be portable, some of the numeric parameters used by an implementation may have hardware dependencies.

1.3 Conformance

1.3.1 Implementation Conformance

1.3.1.3 Conforming Implementation Options

⇒ **1.3.1.3 Conforming Implementation Options** *Add the following to the table of implementation options that warrant requirement by applications or in specifications:*

{_POSIX_ADVISORY_INFO}	Advisory Information option (in 2.9.3)
{_POSIX_CPUTIME}	Process CPU-Time Clocks option (in 2.9.3)
{_POSIX_SPAWN}	Spawn option (in 2.9.3)
{_POSIX_SPORADIC_SERVER}	Process Sporadic Server option (in 2.9.3)
{_POSIX_THREAD_CPUTIME}	Thread CPU-Time Clocks option (in 2.9.3)
{_POSIX_THREAD_SPORADIC_SERVER}	Thread Sporadic Server option (in 2.9.3)
{_POSIX_TIMEOUTS}	Timeouts option (in 2.9.3)

Section 2: Terminology and General Requirements

2.2 Definitions

2.2.2 General Terms

⇒ **2.2.2 General Terms** *Modify the contents of 2.2.2 to add the following definitions in the correct sorted order (disregarding the subclause numbers shown here).*

2.2.2.1 CPU time [execution time]: The time spent executing a process or thread, including the time spent executing system services on behalf of that process or thread. If the Threads option is supported, then the value of the CPU-time clock for a process is implementation defined. With this definition the sum of all the execution times of all the threads in a process might not equal the process execution time, even in a single-threaded process, because implementations may differ in how they account for time during context switches or for other reasons.

2.2.2.2 CPU-time clock: A clock that measures the execution time of a particular process or thread.

2.2.2.3 CPU-time timer: A timer attached to a CPU-time clock.

2.2.2.4 execution time: See *CPU time* in 2.2.2.1.

2.2.3 Abbreviations

For this standard, the following abbreviations apply:

2.2.3.1 C Standard: ISO/IEC 9899: 1995, *Information technology—Programming languages—C.*

2.2.3.2 POSIX.1: ISO/IEC 9945-1: 1996, (IEEE Std 1003.1-1996), *Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API) [C Language].*

2.2.3.3 POSIX.1b: IEEE Std 1003.1b-1993, *Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API)—Amendment b: Realtime Extensions [C Language]*, as amended by *IEEE Std 1003.1i-1995, Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API)—Amendment i: Technical Corrigenda to Realtime Extension [C Language].*

2.2.3.4 POSIX.1c: IEEE Std 1003.1c-1995, *Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API)—Amendment c: Threads Extension [C Language].*

2.2.3.5 POSIX.1d: IEEE Std 1003.1d-1999, *this standard.*

2.2.3.6 POSIX.5 ISO/IEC 14519:1998 {B1}[1)], *POSIX® Ada Language Interfaces—Binding for System Application Program Interfaces (API) including Realtime Extensions.* (This standard includes IEEE Std 1003.5-1992 and IEEE Std 1003.5b-1996.)

2.3 General Concepts

⇒ **2.3 General Concepts—measurement of execution time:** *Add the following subclause, in the proper order, to the existing items in 2.3:*

2.3.1 measurement of execution time: The mechanism used to measure execution time shall be implementation defined. The implementation shall also define to whom will be charged the CPU time that is consumed by interrupt handlers and system services on behalf of the operating system. Execution or CPU time is defined in 2.2.2.1.

---

1) The numbers in curly brackets, when preceded by a “B”, correspond to the numbers of the bibliography in Annex A.

2.7 C Language Definitions

2.7.3 Headers and Function Prototypes

⇒ **2.7.3 Headers and Function Prototypes** *Add the following text after the sentence* "For other functions in this part of ISO/IEC 9945, the prototypes or declarations shall appear in the headers listed below.*":*

Presence of some prototypes or declarations is dependent on implementation options. Where an implementation option is not supported, the prototype or declaration need not be found in the header.

⇒ **2.7.3 Headers and Function Prototypes** *Modify the contents of subclause 2.7.3 to add the following optional headers and functions, at the end of the current list of headers and functions.*

If the Advisory Information option is supported:

`<fcntl.h>` *posix_fadvise*(), *posix_madvise*(), *posix_fallocate*()

If the Message Passing option and the Timeouts option are supported:

`<mqueue.h>` *mq_timedsend*(), *mq_timedreceive*()

If the Thread CPU-Time Clocks option is supported:

`<pthread.h>` *pthread_getcpuclockid*()

If the Threads option and the Timeouts option are supported:

`<pthread.h>` *pthread_mutex_timedlock*()

If the Semaphores option and the Timeouts option are supported:

`<semaphore.h>` *sem_timedwait*()

If the Spawn option is supported:

`<spawn.h>` *posix_spawn*(), *posix_spawnp*(),
posix_spawn_file_actions_init(),
posix_spawn_file_actions_destroy(),
posix_spawn_file_actions_addclose(),
posix_spawn_file_actions_adddup2(),
posix_spawn_file_actions_addopen(),
posix_spawnattr_init(), *posix_spawnattr_destroy*(),
posix_spawnattr_getflags(), *posix_spawnattr_setflags*(),
posix_spawnattr_getpgroup(),
posix_spawnattr_setpgroup(),
posix_spawnattr_getsigmask(),
posix_spawnattr_setsigmask(),
posix_spawnattr_getsigdefault(),
posix_spawnattr_setsigdefault()

If the Spawn option and the Process Scheduling option are supported:

`<spawn.h>`	*posix_spawnattr_getschedpolicy*(), *posix_spawnattr_setschedpolicy*(), *posix_spawnattr_getschedparam*(), *posix_spawnattr_setschedparam*()

If the Advisory Information option is supported:

`<stdlib.h>`	*posix_memalign*()

If the Process CPU-Time Clocks option is supported:

`<time.h>`	*clock_getcpuclockid*()

2.8 Numerical Limits

2.8.2 Minimum Values

⇒ **2.8.2 Minimum Values** *Add the following text after the sentence starting "The symbols in Table 2-3 shall be defined in..."*

The symbols in Table 2-3a shall be defined in `<limits.h>` with the values shown if the associated option is supported.

⇒ **2.8.2 Minimum Values** *Add* Table 2-3a, *described below, after Table 2-3.*

Table 2-3a – Optional Minimum Values

Name	Description	Value	Option
{_POSIX_SS_REPL_MAX}	The number of replenishment operations that may be simultaneously pending for a particular sporadic server scheduler.	4	Process Sporadic Server or Thread Sporadic Server

2.8.4 Run-Time Invariant Values (Possibly Indeterminate)

⇒ **2.8.4 Run-Time Invariant Values (Possibly Indeterminate)** *Replace the whole subclause by the following text:*

The symbols that appear in Table 2-5 that have determinate values shall be defined in `<limits.h>`. The symbols that appear in Table 2-5a that have determinate values shall be defined in `<limits.h>` if the associated option is supported. If any of the values in Table 2-5 or Table 2-5a has a value that is greater than or equal to the stated minimum, but is indeterminate, a definition for that value shall not be defined in `<limits.h>`.

This indetermination might depend on the amount of available memory space on a specific instance of a specific implementation. For the values defined in

Table 2-5, the actual value supported by a specific instance shall be provided by the *sysconf*() function. For the values defined in Table 2-5a, the actual value supported by a specific instance shall be provided by the *sysconf*() function if the associated option is supported.

⇒ **2.8.4 Run-Time Invariant Values (Possibly Indeterminate)** *Add* Table 2-5a, *described next, after Table 2-5.*

Table 2-5a – Optional Run-Time Invariant Values (Possibly Indeterminate)

Name	Description	Minimum Value	Option
{SS_REPL_MAX}	The maximum number of replenishment operations that may be simultaneously pending for a particular sporadic server scheduler.	{_POSIX_SS_-REPL_MAX}	Process Sporadic Server or Thread Sporadic Server

2.8.5 Pathname Variable Values

⇒ **2.8.5 Pathname Variable Values** *Replace the reference to Table 2-6 in the first paragraph of this subclause by:*

Table 2-6 or Table 2-6a

⇒ **2.8.5 Pathname Variable Values** *Replace the sentence* "The actual value supported for a specific pathname shall be provided by the *pathconf*() function" *with the following text:*

For the values defined in Table 2-6, the actual value supported for a specific pathname shall be provided by the *pathconf*() function. For the values defined in Table 2-6a, the actual value supported for a specific pathname shall be provided by the *pathconf*() function if the associated option is supported.

⇒ **2.8.5 Pathname Variable Values** *Add* Table 2-6a, *described next, after Table 2-6.*

Table 2-6a – Optional Pathname Variable Values

Name	Description	Minimum Values	Option
{POSIX_REC_INCR_XFER_SIZE}	Recommended increment for file transfer sizes between the {POSIX_-REC_MIN_XFER_SIZE} and {POSIX_REC_MAX_-XFER_SIZE} values.	*not specified*	Advisory Information
{POSIX_ALLOC_SIZE_MIN}	Minimum number of bytes of storage actually allocated for any portion of a file.	*not specified*	Advisory Information
{POSIX_REC_MAX_XFER_SIZE}	Maximum recommended file transfer size.	*not specified*	Advisory Information
{POSIX_REC_MIN_XFER_SIZE}	Minimum recommended file transfer size.	*not specified*	Advisory Information
{POSIX_REC_XFER_ALIGN}	Recommended file transfer buffer alignment.	*not specified*	Advisory Information

2.9 Symbolic Constants

2.9.3 Compile-Time Symbolic Constants for Portability Specifications

⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**
Change the first words in the first paragraph, currently saying “The constants in Table 2-10 may be used...” *to the following:*

The constants in Table 2-10 and Table 2-10a may be used...

⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**
Add the following sentence at the end of the first paragraph:

If any of the constants in Table 2-10a is defined, it shall be defined with the value shown in that table. This value represents the version of the associated option that is supported by the implementation.

⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**
Add Table 2-10a*, shown below, after Table 2-10.*

Table 2-10a – Versioned Compile-Time Symbolic Constants

Name	Value	Description
{_POSIX_ADVISORY_INFO}	199909L	If this symbol is defined, the implementation supports the Advisory Information option.
{_POSIX_CPUTIME}	199909L	If this symbol is defined, the implementation supports the Process CPU-Time Clocks option.
{_POSIX_SPAWN}	199909L	If this symbol is defined, the implementation supports the Spawn option.
{_POSIX_SPORADIC_SERVER}	199909L	If this symbol is defined, the implementation supports the Process Sporadic Server option.
{_POSIX_THREAD_CPUTIME}	199909L	If this symbol is defined, the implementation supports the Thread CPU-Time Clocks option.
{_POSIX_THREAD_SPORADIC_SERVER}	199909L	If this symbol is defined, the implementation supports the Thread Sporadic Server option.
{_POSIX_TIMEOUTS}	199909L	If this symbol is defined, the implementation supports the Timeouts option.

⇒ 2.9.3 Compile-Time Symbolic Constants for Portability Specifications

Add the following paragraphs before the last paragraph in 2.9.3:

If the symbol {_POSIX_SPORADIC_SERVER} is defined, then the symbol {_POSIX_PRIORITY_SCHEDULING} shall also be defined. If the symbol {_POSIX_THREAD_SPORADIC_SERVER} is defined, then the symbol {_POSIX_-THREAD_PRIORITY_SCHEDULING} shall also be defined.

If the symbol {_POSIX_CPUTIME} is defined, then the symbol {_POSIX_TIMERS} shall also be defined. If the symbol {_POSIX_THREAD_CPUTIME} is defined, then the symbol {_POSIX_TIMERS} shall also be defined.

Section 3: Process Primitives

3.1 Process Creation and Execution

3.1.1 Process Creation

3.1.1.2 Description

⇒ **3.1.1.2 Process Creation—Description** *Add the following paragraphs to the description of the* fork*() function:*

If {_POSIX_CPUTIME} is defined:

> The initial value of the CPU-time clock of the child process shall be set to zero.

If {_POSIX_THREAD_CPUTIME} is defined:

> The initial value of the CPU-time clock of the single thread of the child process shall be set to zero.

3.1.2 Execute a File

3.1.2.2 Description

⇒ **3.1.2.2 Execute a File—Description** *Add the following paragraph to the description of the family of* exec *functions.*

If {_POSIX_CPUTIME} is defined:

> The new process image shall inherit the CPU-time clock of the calling process image. This inheritance means that the process CPU-time clock of the process being *exec*ed shall not be reinitialized or altered as a result of the *exec* function other than to reflect the time spent by the process executing the *exec* function itself.

If {_POSIX_THREAD_CPUTIME} is defined:

> The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero.

⇒ **3.1 Process Creation and Execution** *Add the following subclauses:*

3.1.4 Spawn File Actions

Functions: *posix_spawn_file_actions_init*(), *posix_spawn_file_actions_destroy*(), *posix_spawn_file_actions_addclose*(), *posix_spawn_file_actions_adddup2*(), *posix_spawn_file_actions_addopen*().

3.1.4.1 Synopsis

```
#include <sys/types.h>
#include <spawn.h>

int posix_spawn_file_actions_init(
          posix_spawn_file_actions_t *file_actions);

int posix_spawn_file_actions_destroy(
          posix_spawn_file_actions_t *file_actions);

int posix_spawn_file_actions_addclose(
          posix_spawn_file_actions_t *file_actions,
          int fildes);

int posix_spawn_file_actions_adddup2(
          posix_spawn_file_actions_t *file_actions,
          int fildes, int newfildes);

int posix_spawn_file_actions_addopen(
          posix_spawn_file_actions_t *file_actions,
          int fildes, const char *path,
          int oflag, mode_t mode);
```

3.1.4.2 Description

If {_POSIX_SPAWN} is defined:

A spawn file actions object is of type *posix_spawn_file_actions_t* (defined in `<spawn.h>`) and is used to specify a series of actions to be performed by a *posix_spawn*() or *posix_spawnp*() operation in order to arrive at the set of open file descriptors for the child process given the set of open file descriptors of the parent. This standard does not define comparison or assignment operators for the type *posix_spawn_file_actions_t*.

The *posix_spawn_file_actions_init*() function initializes the object referenced by *file_actions* to contain no file actions for *posix_spawn*() or *posix_spawnp*() to perform.

The effect of initializing an already initialized spawn file actions object is undefined.

The *posix_spawn_file_actions_destroy*() function destroys the object referenced by *file_actions*; the object becomes, in effect, uninitialized. An implementation may cause *posix_spawn_file_actions_destroy*() to set the object referenced by *file_actions* to an invalid value. A destroyed spawn file actions object can be reinitialized using *posix_spawn_file_actions_init*(); the results of otherwise referencing the object after it has been destroyed are undefined.

The *posix_spawn_file_actions_addclose*() function adds a close action to the object referenced by *file_actions* that will cause the file descriptor *fildes* to be closed [as if *close*(*fildes*) had been called] when a new process is spawned using this file actions object.

The *posix_spawn_file_actions_adddup2*() function adds a dup2 action to the object referenced by *file_actions* that will cause the file descriptor *fildes* to be duplicated as *newfildes* [as if *dup2*(*fildes*, *newfildes*) had been called] when a new process is spawned using this file actions object.

The *posix_spawn_file_actions_addopen*() function adds an open action to the object referenced by *file_actions* that will cause the file named by *path* to be opened [as if *open*(*path*, *oflag*, *mode*) had been called, and the returned file descriptor, if not *fildes*, had been changed to *fildes*] when a new process is spawned using this file actions object. If *fildes* was already an open file descriptor, it shall be closed before the new file is opened.

A spawn file actions object, when passed to *posix_spawn*() or *posix_spawnp*(), shall specify how the set of open file descriptors in the calling process is transformed into a set of potentially open file descriptors for the spawned process. This transformation shall be as if the specified sequence of actions was performed exactly once, in the context of the spawned process (prior to execution of the new process image), in the order in which the actions were added to the object; additionally, when the new process image is executed, any file descriptor (from this new set) which has its FD_CLOEXEC flag set will be closed (see 3.1.6).

Otherwise:

Either the implementation shall support the *posix_spawn_file_actions_init*(), *posix_spawn_file_actions_destroy*(), *posix_spawn_file_actions_addclose*(), *posix_spawn_file_actions_adddup2*(), and *posix_spawn_file_actions_addopen*() functions as described above, or these functions shall not be provided.

3.1.4.3 Returns

Upon successful completion, the *posix_spawn_file_actions_init*(), *posix_spawn_file_actions_destroy*(), *posix_spawn_file_actions_addclose*(), *posix_spawn_file_actions_adddup2*(), or *posix_spawn_file_actions_addopen*() operation shall return zero. Otherwise, an error number shall be returned to indicate the error.

3.1.4.4 Errors

For each of the following conditions, if the condition is detected, the *posix_spawn_file_actions_init*(), *posix_spawn_file_actions_addclose*(), *posix_spawn_file_actions_adddup2*(), or *posix_spawn_file_actions_addopen*() function shall return the corresponding error number:

[ENOMEM] Insufficient memory exists to initialize or add to the spawn file actions object.

For each of the following conditions, if the condition is detected, the *posix_spawn_file_actions_destroy*(), *posix_spawn_file_actions_addclose*(), *posix_spawn_file_actions_adddup2*(), or *posix_spawn_file_actions_addopen*() function shall return the corresponding error number:

[EINVAL] The value specified by *file_actions* is invalid.

For each of the following conditions, the *posix_spawn_file_actions_addclose*(), *posix_spawn_file_actions_adddup2*(), or *posix_spawn_file_actions_addopen*() function shall return the corresponding error number:

[EBADF] The value specified by *fildes* is negative or greater than or equal to {OPEN_MAX}.

It shall not be considered an error for the *fildes* argument passed to the *posix_spawn_file_actions_addclose*(), *posix_spawn_file_actions_adddup2*(), or *posix_spawn_file_actions_addopen*() functions to specify a file descriptor for which the specified operation could not be performed at the time of the call. Any such error will be detected when the associated file actions object is later used during a *posix_spawn*() or *posix_spawnp*() operation.

3.1.4.5 Cross-References

close(), 6.3.1; *dup2*(), 6.2.1; *open*(), 5.3.1; *posix_spawn*(), 3.1.6; *posix_spawnp*(), 3.1.6;

3.1.5 Spawn Attributes

Functions: *posix_spawnattr_init*(), *posix_spawnattr_destroy*(), *posix_spawnattr_getflags*(), *posix_spawnattr_setflags*(), *posix_spawnattr_getpgroup*(), *posix_spawnattr_setpgroup*(), *posix_spawnattr_getschedpolicy*(), *posix_spawnattr_setschedpolicy*(), *posix_spawnattr_getschedparam*(), *posix_spawnattr_setschedparam*(), *posix_spawnattr_getsigmask*(), *posix_spawnattr_setsigmask*(), *posix_spawnattr_getsigdefault*(), *posix_spawnattr_setsigdefault*().

3.1.5.1 Synopsis

```
#include <sys/types.h>
#include <signal.h>
#include <spawn.h>

int posix_spawnattr_init (posix_spawnattr_t *attr);

int posix_spawnattr_destroy (posix_spawnattr_t *attr);

int posix_spawnattr_getflags (const posix_spawnattr_t *attr,
                    short *flags);

int posix_spawnattr_setflags (posix_spawnattr_t *attr,
                    short flags);

int posix_spawnattr_getpgroup (const posix_spawnattr_t *attr,
                    pid_t *pgroup);

int posix_spawnattr_setpgroup (posix_spawnattr_t *attr,
                    pid_t pgroup);

int posix_spawnattr_getsigmask (const posix_spawnattr_t *attr,
                    sigset_t *sigmask);

int posix_spawnattr_setsigmask (posix_spawnattr_t *attr,
                    const sigset_t *sigmask);

int posix_spawnattr_getsigdefault (const posix_spawnattr_t *attr,
                    sigset_t *sigdefault);

int posix_spawnattr_setsigdefault (posix_spawnattr_t *attr,
                    const sigset_t *sigdefault);


#include <sched.h>

int posix_spawnattr_getschedpolicy (const posix_spawnattr_t *attr,
                    int *schedpolicy);

int posix_spawnattr_setschedpolicy (posix_spawnattr_t *attr,
                    int schedpolicy);

int posix_spawnattr_getschedparam (const posix_spawnattr_t *attr,
                    struct sched_param *schedparam);

int posix_spawnattr_setschedparam (posix_spawnattr_t *attr,
                    const struct sched_param *schedparam);
```

3.1.5.2 Description

If {_POSIX_SPAWN} is defined:

A spawn attributes object is of type *posix_spawnattr_t* (defined in `<spawn.h>`) and is used to specify the inheritance of process attributes across a spawn operation. This standard does not define comparison or assignment operators for the type *posix_spawnattr_t*.

The function *posix_spawnattr_init*() initializes a spawn attributes object *attr* with the default value for all of the individual attributes used by the implementation.

Each implementation shall document the individual attributes it uses and their default values unless these values are defined by this standard.

The resulting spawn attributes object (possibly modified by setting individual attribute values) is used to modify the behavior of *posix_spawn*() or *posix_spawnp*() (see 3.1.6). After a spawn attributes object has been used to spawn a process by a call to a *posix_spawn*() or *posix_spawnp*(), any function affecting the attributes object (including destruction) does not affect any process that has been spawned in this way.

The *posix_spawnattr_destroy*() function destroys a spawn attributes object. The effect of subsequent use of the object is undefined until the object is re-initialized by another call to *posix_spawnattr_init*(). An implementation may cause *posix_spawnattr_destroy*() to set the object referenced by *attr* to an invalid value.

The **spawn-flags** attribute is used to indicate which process attributes are to be changed in the new process image when invoking *posix_spawn*() or *posix_spawnp*(). It is the inclusive OR of zero or more of the flags POSIX_SPAWN_SETPGROUP, POSIX_SPAWN_RESETIDS, POSIX_SPAWN_SETSIGMASK, and POSIX_SPAWN_SETSIGDEF. In addition, if the Process Scheduling option is supported, the flags POSIX_SPAWN_SETSCHEDULER and POSIX_SPAWN_SETSCHEDPARAM shall also be supported. These flags are defined in **<spawn.h>**. The default value of this attribute shall be with no flags set.

The *posix_spawnattr_setflags*() function is used to set the **spawn-flags** attribute in an initialized attributes object referenced by *attr*. The *posix_spawnattr_getflags*() function obtains the value of the **spawn-flags** attribute from the attributes object referenced by *attr*.

The **spawn-pgroup** attribute represents the process group to be joined by the new process image in a spawn operation (if POSIX_SPAWN_SETPGROUP is set in the **spawn-flags** attribute). The default value of this attribute shall be zero.

The *posix_spawnattr_setpgroup*() function is used to set the **spawn-pgroup** attribute in an initialized attributes object referenced by *attr*. The *posix_spawnattr_getpgroup*() function obtains the value of the **spawn-pgroup** attribute from the attributes object referenced by *attr*.

The **spawn-sigmask** attribute represents the signal mask in effect in the new process image of a spawn operation (if POSIX_SPAWN_SETSIGMASK is set in the **spawn-flags** attribute). The default value of this attribute is unspecified.

The *posix_spawnattr_setsigmask*() function is used to set the **spawn-sigmask** attribute in an initialized attributes object referenced by *attr*. The *posix_spawnattr_getsigmask*() function obtains the value of the **spawn-sigmask** attribute from the attributes object referenced by *attr*.

The **spawn-sigdefault** attribute represents the set of signals to be forced to default signal handling in the new process image (if POSIX_SPAWN_SETSIGDEF is set in the **spawn-flags** attribute). The default value of this attribute shall be an empty signal set.

The *posix_spawnattr_setsigdefault*() function is used to set the **spawn-sigdefault** attribute in an initialized attributes object referenced by *attr*. The *posix_spawnattr_getsigdefault*() function obtains the value of the **spawn-sigdefault** attribute from the attributes object referenced by *attr*.

Otherwise:

Either the implementation shall support the *posix_spawnattr_init*(), *posix_spawnattr_destroy*(), *posix_spawnattr_getflags*(), *posix_spawnattr_setflags*(), *posix_spawnattr_getpgroup*(), *posix_spawnattr_setpgroup*(), *posix_spawnattr_getsigmask*(), *posix_spawnattr_setsigmask*(), *posix_spawnattr_getsigdefault*(), and *posix_spawnattr_setsigdefault*() functions as described above, or these functions shall not be provided.

If {_POSIX_SPAWN} and {_POSIX_PRIORITY_SCHEDULING} are both defined:

The **spawn-schedpolicy** attribute represents the scheduling policy to be assigned to the new process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER is set in the **spawn-flags** attribute). The default value of this attribute is unspecified.

The *posix_spawnattr_setschedpolicy*() function is used to set the **spawn-schedpolicy** attribute in an initialized attributes object referenced by *attr*. The *posix_spawnattr_getschedpolicy*() function obtains the value of the **spawn-schedpolicy** attribute from the attributes object referenced by *attr*.

The **spawn-schedparam** attribute represents the scheduling parameters to be assigned to the new process image in a spawn operation (if POSIX_SPAWN_SETSCHEDULER or POSIX_SPAWN_SETSCHEDPARAM is set in the **spawn-flags** attribute). The default value of this attribute is unspecified.

The *posix_spawnattr_setschedparam*() function is used to set the **spawn-schedparam** attribute in an initialized attributes object referenced by *attr*. The *posix_spawnattr_getschedparam*() function obtains the value of the **spawn-schedparam** attribute from the attributes object referenced by *attr*.

Otherwise:

Either the implementation shall support the *posix_spawnattr_getschedpolicy*(), *posix_spawnattr_setschedpolicy*(), *posix_spawnattr_getschedparam*(), and *posix_spawnattr_setschedparam*() functions as described above, or these functions shall not be provided.

Additional attributes, their default values, and the names of the associated functions to get and set those attribute values are implementation defined.

3.1.5.3 Returns

If successful, the *posix_spawnattr_init*(), *posix_spawnattr_destroy*(), *posix_spawnattr_setflags*(), *posix_spawnattr_setpgroup*(), *posix_spawnattr_setschedpolicy*(), *posix_spawnattr_setschedparam*(), *posix_spawnattr_setsigmask*(), and *posix_spawnattr_setsigdefault*() functions shall return zero. Otherwise, an error number shall be returned to indicate the error.

If successful, the *posix_spawnattr_getflags*(), *posix_spawnattr_getpgroup*(), *posix_spawnattr_getschedpolicy*(), *posix_spawnattr_getschedparam*(), *posix_spawnattr_getsigmask*(), and *posix_spawnattr_getsigdefault*() functions shall return zero and store the value of the `spawn-flags`, `spawn-pgroup`, `spawn-schedpolicy`, `spawn-schedparam`, `spawn-sigmask`, or `spawn-sigdefault` attribute of *attr* into the object referenced by the *flags*, *pgroup*, *schedpolicy*, *schedparam*, *sigmask*, or *sigdefault* parameter, respectively. Otherwise, an error number shall be returned to indicate the error.

3.1.5.4 Errors

If any of the following conditions occur, the *posix_spawnattr_init*() function shall return the corresponding error value:

[ENOMEM] Insufficient memory exists to initialize the spawn attributes object.

For each of the following conditions, if the condition is detected, the *posix_spawnattr_destroy*(), *posix_spawnattr_getflags*(), *posix_spawnattr_setflags*(), *posix_spawnattr_getpgroup*(), *posix_spawnattr_setpgroup*(), *posix_spawnattr_getschedpolicy*(), *posix_spawnattr_setschedpolicy*(), *posix_spawnattr_getschedparam*(), *posix_spawnattr_setschedparam*(), *posix_spawnattr_getsigmask*(), *posix_spawnattr_setsigmask*(), *posix_spawnattr_getsigdefault*(), and *posix_spawnattr_setsigdefault*() functions shall return the corresponding error value:

[EINVAL] The value specified by *attr* is invalid.

For each of the following conditions, if the condition is detected, the *posix_spawnattr_setflags*(), *posix_spawnattr_setpgroup*(), *posix_spawnattr_setschedpolicy*(), *posix_spawnattr_setschedparam*(), *posix_spawnattr_setsigmask*(), and *posix_spawnattr_setsigdefault*() functions shall return the corresponding error value:

[EINVAL] The value of the attribute being set is not valid.

3.1.5.5 Cross-References

posix_spawn(), 3.1.6; *posix_spawnp*(), 3.1.6.

3.1.6 Spawn a Process

Functions: *posix_spawn*(), *posix_spawnp*().

3.1.6.1 Synopsis

```
#include <sys/types.h>
#include <spawn.h>

int posix_spawn(pid_t *pid,
          const char *path,
          const posix_spawn_file_actions_t *file_actions,
          const posix_spawnattr_t *attrp,
          char * const argv[],
          char * const envp[]);

int posix_spawnp(pid_t *pid,
          const char *file,
          const posix_spawn_file_actions_t *file_actions,
          const posix_spawnattr_t *attrp,
          char * const argv[],
          char * const envp[]);
```

3.1.6.2 Description

If {_POSIX_SPAWN} is defined:

The *posix_spawn*() and *posix_spawnp*() functions shall create a new process (child process) from the specified process image. The new process image is constructed from a regular executable file called the new process image file.

When a C program is executed as the result of this call, it shall be entered as a C language function call as follows:

```
int main (int argc, char *argv[]);
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the variable

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings.

The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array shall be a **NULL** pointer and is not counted in *argc*. These strings constitute the argument list available to the new process image. The value in *argv[0]* should point to a filename that is associated with the process image being started by the *posix_spawn*() or *posix_spawnp*() function.

The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a **NULL** pointer.

The number of bytes available for the child process's combined argument and environment lists is {ARG_MAX}. The implementation shall specify in the system documentation (see 1.3.1) whether any list overhead, such as length words, null terminators, pointers, or alignment bytes, is included in this total.

The *path* argument to *posix_spawn*() is a pathname that identifies the new process image file to execute.

The *file* parameter to *posix_spawnp*() shall be used to construct a pathname that identifies the new process image file. If the *file* parameter contains a slash character, the *file* parameter shall be used as the pathname for the new process image file. Otherwise, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable **PATH** (see 2.6). If this environment variable is not defined, the results of the search are implementation defined.

If *file_actions* is a **NULL** pointer, then file descriptors open in the calling process shall remain open in the child process, except for those whose close-on-exec flag FD_CLOEXEC is set (see 6.5.2 and 6.5.1). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see 6.5.2), shall remain unchanged.

If *file_actions* is not **NULL**, then the file descriptors open in the child process shall be those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions shall be

1. The set of open file descriptors for the child process shall initially be the same set as is open for the calling process. All attributes of the corresponding open file descriptions, including file locks (see 6.5.2), shall remain unchanged.

2. The signal mask and the effective user and group IDs for the child process shall be changed as specified in the attributes object referenced by *attrp*.

3. The file actions specified by the spawn file actions object shall be performed in the order in which they were added to the spawn file actions object.

4. Any file descriptor that has its FD_CLOEXEC flag set (see 6.5.2) shall be closed.

The *posix_spawnattr_t* spawn attributes object type is defined in `<spawn.h>`. It shall contain at least the attributes described in 3.1.5.

If the POSIX_SPAWN_SETPGROUP flag is set in the `spawn-flags` attribute of the object referenced by *attrp* and the `spawn-pgroup` attribute of the same object is non-zero, then the child's process group shall be as specified in the `spawn-pgroup` attribute of the object referenced by *attrp*.

As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the `spawn-flags` attribute of the object referenced by *attrp* and the `spawn-pgroup` attribute of the same object is set to zero, then the child shall be in a new process group with a process group ID equal to its process ID.

If the POSIX_SPAWN_SETPGROUP flag is not set in the `spawn-flags` attribute of the object referenced by *attrp*, the new child process shall inherit the parent's process group.

If {_POSIX_PRIORITY_SCHEDULING} is defined and the POSIX_SPAWN_SETSCHEDPARAM flag is set in the `spawn-flags` attribute of the object referenced by *attrp*, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image shall initially have the scheduling policy of the calling process with the scheduling parameters specified in the `spawn-`

`schedparam` attribute of the object referenced by *attrp*.

If {_POSIX_PRIORITY_SCHEDULING} is defined and the POSIX_SPAWN_SETSCHEDULER flag is set in `spawn-flags` attribute of the object referenced by *attrp* (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag), the new process image shall initially have the scheduling policy specified in the `spawn-schedpolicy` attribute of the object referenced by *attrp* and the scheduling parameters specified in the `spawn-schedparam` attribute of the same object.

The POSIX_SPAWN_RESETIDS flag in the `spawn-flags` attribute of the object referenced by *attrp* governs the effective user ID of the child process. If this flag is not set, the child process inherits the parent process's effective user ID. If this flag is set, the child process's effective user ID is reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process will become that file's owner ID before the new process image begins execution.

The POSIX_SPAWN_RESETIDS flag in the `spawn-flags` attribute of the object referenced by *attrp* also governs the effective group ID of the child process. If this flag is not set, the child process inherits the parent process's effective group ID. If this flag is set, the child process's effective group ID is reset to the parent's real group ID. In either case, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the child process will become that file's group ID before the new process image begins execution.

If the POSIX_SPAWN_SETSIGMASK flag is set in the `spawn-flags` attribute of the object referenced by *attrp*, the child process shall initially have the signal mask specified in the `spawn-sigmask` attribute of the object referenced by *attrp*.

If the POSIX_SPAWN_SETSIGDEF flag is set in the `spawn-flags` attribute of the object referenced by *attrp*, the signals specified in the `spawn-sigdefault` attribute of the same object shall be set to their default actions in the child process. Signals set to their default actions in the parent process shall be set to their default actions in the child process.

Signals set to be caught by the calling process shall be set to their default actions in the child process.

Signals set to be ignored by the calling process image shall be set to be ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the `spawn-flags` attribute of the object referenced by *attrp* and the signals being indicated in the `spawn-sigdefault` attribute of the object referenced by *attrp*.

If the value of the *attrp* pointer is **NULL**, then the default values are used.

All process attributes other than those influenced by the attributes set in the object referenced by *attrp* as specified above or by the file descriptor manipulations specified in *file_actions* shall appear in the new process image as though *fork*() had been called to create a child process and then a member of the *exec* family of functions had been called by the child process to execute the new process image.

If the Threads option is supported, then it is implementation defined whether the fork handlers are run when *posix_spawn*() or *posix_spawnp*() is called.

Otherwise :

Either the implementation shall support the *posix_spawn*() and *posix_spawnp*() functions as described above, or these functions shall not be provided.

3.1.6.3 Returns

Upon successful completion, the *posix_spawn*() or *posix_spawnp*() operation shall return the process ID of the child process to the parent process, in the variable pointed to by a non-**NULL** *pid* argument, and shall return zero as the function return value. Otherwise, no child process shall be created, the value stored into the variable pointed to by a non-**NULL** *pid* is unspecified, and the corresponding error value shall be returned as the function return value. If the *pid* argument is the **NULL** pointer, the process ID of the child is not returned to the caller.

3.1.6.4 Errors

For each of the following conditions, if the condition is detected, the *posix_spawn*() or *posix_spawnp*() function shall fail and post the corresponding status value or, if the error occurs after the calling process successfully returns from the *posix_spawn*() or *posix_spawnp*() function, shall cause the child process to exit with exit status 127:

[EINVAL] The value specified by *file_actions* or *attrp* is invalid.

If *posix_spawn*() or *posix_spawnp*() fails for any of the reasons that would cause *fork*() or one of the *exec* family of functions to fail, an error value shall be returned as described by *fork*() and *exec*, respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If POSIX_SPAWN_SETPGROUP is set in the `spawn-flags` attribute of the object referenced by *attrp* and *posix_spawn*() or *posix_spawnp*() fails while changing the child's process group, an error value shall be returned as described by *setpgid*() (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If {_POSIX_PRIORITY_SCHEDULING} is defined, if POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set in the `spawn-flags` attribute of the object referenced by *attrp*, and if *posix_spawn*() or *posix_spawnp*() fails for any of the reasons that would cause *sched_setparam*() to fail, an error value shall be returned as described by *sched_setparam*() (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If {_POSIX_PRIORITY_SCHEDULING} is defined, if POSIX_SPAWN_SETSCHEDULER is set in the `spawn-flags` attribute of the object referenced by *attrp*, and if *posix_spawn*() or *posix_spawnp*() fails for any of the reasons that would cause *sched_setscheduler*() to fail, an error value shall be returned as described by *sched_setscheduler*() (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127).

If the *file_actions* argument is not **NULL** and specifies any close, dup2, or open actions to be performed and if *posix_spawn*() or *posix_spawnp*() fails for any of the reasons that would cause *close*(), *dup2*(), or *open*() to fail, an error value shall be returned as described by *close*(), *dup2*(), and *open*(), respectively (or, if the error occurs after the calling process successfully returns, the child process exits with exit status 127). An open file action may, by itself, result in any of the errors described by *close*() or *dup2*(), in addition to those described by *open*().

3.1.6.5 Cross-References

alarm(), 3.4.1; *chmod*(), 5.6.4; *close*(), 6.3.1; *dup2*(), 6.2.1; *exec*, 3.1.2; *_exit*(), 3.2.2; *fcntl*(), 6.5.2; *fork*(), 3.1.1; *kill*(), 3.3.2; *open*(), 5.3.1; *posix_spawn_file_actions_init*(), 3.1.4; *posix_spawn_file_actions_destroy*(), 3.1.4; *posix_spawn_file_actions_addclose*(), 3.1.4; *posix_spawn_file_actions_adddup2*(), 3.1.4; *posix_spawn_file_actions_addopen*(), 3.1.4; *posix_spawnattr_init*(), 3.1.5; *posix_spawnattr_destroy*(), 3.1.5; *posix_spawnattr_getflags*(), 3.1.5; *posix_spawnattr_setflags*(), 3.1.5; *posix_spawnattr_getpgroup*(), 3.1.5; *posix_spawnattr_setpgroup*(), 3.1.5; *posix_spawnattr_getschedpolicy*(), 3.1.5; *posix_spawnattr_setschedpolicy*(), 3.1.5; *posix_spawnattr_getschedparam*(), 3.1.5; *posix_spawnattr_setschedparam*(), 3.1.5; *posix_spawnattr_getsigmask*(), 3.1.5; *posix_spawnattr_setsigmask*(), 3.1.5; *posix_spawnattr_getsigdefault*(), 3.1.5; *posix_spawnattr_setsigdefault*(), 3.1.5; *sched_setparam*(), 13.3.1; *sched_setscheduler*(), 13.3.3; *setpgid*(), 4.3.3; *setuid*(), 4.2.2; *stat*(), 5.6.2; *times*(), 4.5.2; *wait*, 3.2.1.

3.2 Process Termination

3.2.1 Wait for Process Termination

3.2.1.2 Wait for Process Termination — Description

⇒ **3.2.1.2 Wait for Process Termination — Description** *Add the following paragraphs after the definition of the WSTOPSIG(stat_val) macro:*

It is unspecified whether the status value returned by calls to *wait*() or *waitpid*() for processes created by *posix_spawn*() or *posix_spawnp*() may indicate a WIFSTOPPED(*stat_val*) before subsequent calls to *wait*() or *waitpid*() indicate WIFEXITED(*stat_val*) as the result of an error detected before the new process image starts executing.

It is unspecified whether the status value returned by calls to *wait*() or *waitpid*() for processes created by *posix_spawn*() or *posix_spawnp*() may indicate a WIFSIGNALED(*stat_val*) if a signal is sent to the parent's process group after *posix_spawn*() or *posix_spawnp*() is called.

Section 4: Process Environment

4.8 Configurable System Variables

4.8.1 Get Configurable System Variables

4.8.1.2 Description

⇒ **4.8.1.2 Get Configurable System Variables—Description** *Add the following text after the sentence* "The implementation shall support all of the variables listed in Table 4-2 and may support others", *in the second paragraph:*

Support for some configuration variables is dependent on implementation options (see Table 4-3). Where an implementation option is not supported, the variable need not be supported.

⇒ **4.8.1.2 Get Configurable System Variables—Description** *In the second paragraph, replace the text* "The variables in Table 4-2 come from ..." *by the following:*

"The variables in Table 4-2 and Table 4-3 come from ..."

⇒ **4.8.1.2 Get Configurable System Variables—Description** *Add the following table:*

Table 4-3 – Optional Configurable System Variables

Variable	*name* Value	Required Option
{_POSIX_SPAWN}	_SC_SPAWN	Spawn
{_POSIX_TIMEOUTS}	_SC_TIMEOUTS	Timeouts
{_POSIX_CPUTIME}	_SC_CPUTIME	Process CPU-Time Clocks
{_POSIX_THREAD_CPUTIME}	_SC_THREAD_CPUTIME	Thread CPU-Time Clocks
{_POSIX_SPORADIC_SERVER}	_SC_SPORADIC_SERVER	Process Sporadic Server
{_POSIX_THREAD_SPORADIC_SERVER}	_SC_THREAD_SPORADIC_SERVER	Thread Sporadic Server
{_POSIX_ADVISORY_INFO}	_SC_ADVISORY_INFO	Advisory Information

Section 5: Files and Directories

5.7 Configurable Pathname Variables

5.7.1 Get Configurable Pathname Variables

5.7.1.2 Description

⇒ **5.7.1.2 Get Configurable Pathname Variables—Description** *Add the following text after the sentence* "The implementation shall support all of the variables listed in Table 5-2 and may support others", *in the third paragraph:*

Support for some pathname configuration variables is dependent on implementation options (see Table 5-3). Where an implementation option is not supported, the variable need not be supported.

⇒ **5.7.1.2 Get Configurable Pathname Variables—Description** *In the third paragraph, replace the text* "The variables in Table 5-2 come from ..." *by the following:*

"The variables in Table 5-2 and Table 5-3 come from ..."

⇒ **5.7.1.2 Get Configurable Pathname Variables—Description** *Add the following table:*

Table 5-3 – Optional Configurable Pathname Variables

Variable	***name*** **Value**	Required Option
{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	Advisory Information
{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	Advisory Information
{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	Advisory Information
{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	Advisory Information
{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	Advisory Information

Section 6: Input and Output Primitives

6.7 Asynchronous Input and Output

6.7.1 Data Definitions for Asynchronous Input and Output

6.7.1.1 Asynchronous I/O Control Block

⇒ **6.7.1.1 Asynchronous I/O Control Block** *Change the sentence, in the fifth paragraph, beginning with* "The order of processing of requests submitted by processes whose schedulers . . . " *to the following:*

Unless both {_POSIX_PRIORITIZED_IO} and {_POSIX_PRIORITY_SCHEDULING} are defined, the order of processing asynchronous I/O requests is unspecified. When both {_POSIX_PRIORITIZED_IO} and {_POSIX_PRIORITY_SCHEDULING} are defined, the order of processing of requests submitted by processes whose schedulers are not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC is unspecified.

Section 11: Synchronization

11.2 Semaphore Functions

11.2.6 Lock a Semaphore

⇒ **11.2.6 Lock a Semaphore** *Add the following function at the end of the list of functions:*

sem_timedwait().

11.2.6.1 Synopsis

⇒ **11.2.6.1 Lock a Semaphore—Synopsis** *Add the following #include and prototype at the end of the synopsis:*

```
#include <time.h>
int sem_timedwait(sem_t *sem,
                  const struct timespec *abs_timeout);
```

11.2.6.2 Description

⇒ **11.2.6.2 Lock a Semaphore—Description** *Add the following text at the end of the description:*

If {_POSIX_SEMAPHORES} and {_POSIX_TIMEOUTS} are both defined:

The *sem_timedwait*() function locks the semaphore referenced by *sem* as in the *sem_wait*() function. However, if the semaphore cannot be locked without waiting for another process or thread to unlock the semaphore by performing a *sem_post*() function, this wait shall be terminated when the specified timeout expires.

The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (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. If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock. If the Timers option is not supported, the timeout is based on the system clock as returned by the *time*() function. The resolution of the timeout is the resolution of the clock on which it is based. The *timespec* datatype is defined as a structure in the header `<time.h>`.

Under no circumstance will the function fail with a timeout if the semaphore can be locked immediately. The validity of the *abs_timeout* argument need not be checked if the semaphore can be locked immediately.

Otherwise:

Either the implementation shall support the *sem_timedwait*() function as described above, or this function shall not be provided.

11.2.6.3 Returns

⇒ **11.2.6.3 Lock a Semaphore—Returns** *Add the following function to the list of functions:*

sem_timedwait()

11.2.6.4 Errors

⇒ **11.2.6.4 Lock a Semaphore — Errors** *Make the following changes to the discussion of error conditions:*

Add *sem_timedwait*() to the list of functions for both the standard error conditions and the ‘‘if detected’’ error conditions.

Add an [ETIMEDOUT] error value with the following reason, to the list of errors that must be detected:

The semaphore could not be locked before the specified timeout expired.

To the [EINVAL] error description, add the following reason:

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

11.2.6.5 Cross-References

⇒ **11.2.6.5 Lock a Semaphore—Cross-References** *Add the following items to the cross-references in alphabetical order:*

time(), 4.5.1; `<time.h>`, 14.1.

11.2.7 Unlock a Semaphore

⇒ **11.2.7.2 Unlock a Semaphore—Description** *(The following change is made in a context where the Process Scheduling option is supported.) Change the sentence, beginning with* "In the case of the schedulers . . . " *to the following:*

In the case of the schedulers {SCHED_FIFO}, {SCHED_RR}, and {SCHED_-SPORADIC}, the highest priority waiting thread shall be unblocked, and if there is more than one highest-priority thread blocked waiting for the semaphore, then the highest-priority thread that has been waiting the longest shall be unblocked.

11.3 Mutexes

11.3.3 Locking and Unlocking a Mutex

⇒ **11.3.3 Locking and Unlocking a Mutex** *Add the following function at the end of the list:*

pthread_mutex_timedlock().

11.3.3.1 Synopsis

⇒ **11.3.3.1 Locking and Unlocking a Mutex—Synopsis** *Add the following #include and prototype at the end of the synopsis:*

```
#include <time.h>
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *abs_timeout);
```

11.3.3.2 Description

⇒ **11.3.3.2 Locking and Unlocking a Mutex—Description** *Add the following text at the end of the description:*

If {_POSIX_THREADS} and {_POSIX_TIMEOUTS} are both defined:

> The *pthread_mutex_timedlock*() function is called to lock the mutex object referenced by *mutex*. If the mutex is already locked, the calling thread blocks until the mutex becomes available as in the *pthread_mutex_lock*() function. If the mutex cannot be locked without waiting for another thread to unlock the mutex, this wait shall be terminated when the specified timeout expires.
>
> The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (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. If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout is based on the system clock as returned by the *time*() function. The resolution of the timeout is the resolution of the clock on which it is based. The *timespec* datatype is defined as a structure in the header `<time.h>`.

Under no circumstance will the function fail with a timeout if the mutex can be locked immediately. The validity of the *abs_timeout* parameter need not be checked if the mutex can be locked immediately.

As a consequence of the priority inheritance rules (for mutexes initialized with the PRIO_INHERIT protocol), if a timed mutex wait is terminated because its timeout expires, the priority of the owner of the mutex will be adjusted as necessary to reflect the fact that this thread is no longer among the threads waiting for the mutex.

Otherwise:

Either the implementation shall support the *pthread_mutex_timedlock*() function as described above, or the function shall not be provided.

11.3.3.3 Returns

⇒ **11.3.3.3 Locking and Unlocking a Mutex—Returns** *Add the following function to the list of functions:*

pthread_mutex_timedlock()

11.3.3.4 Errors

⇒ **11.3.3.4 Locking and Unlocking a Mutex—Errors** *Make the following changes to the discussion of error conditions:*

Add *pthread_mutex_timedlock*() to the list of functions for the [EINVAL] and [EDEADLK] conditions.

To the [EINVAL] error description, add the following reason:

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

New paragraph with one error condition: If the following conditions occur, the *pthread_mutex_timedlock*() function shall return the corresponding error number:

[ETIMEDOUT] The mutex could not be locked before the specified timeout expired.

11.3.3.5 Cross-References

⇒ **11.3.3.5 Locking and Unlocking a Mutex—Cross-References** *Add the following items to the cross-references in alphabetical order:*

time(), 4.5.1; `<time.h>`, 14.1.

Section 13: Execution Scheduling

13.1 Scheduling Parameters

⇒ **13.1 Scheduling Parameters** *Add the following paragraph after the first paragraph and associated table:*

In addition, if {_POSIX_SPORADIC_SERVER} or {_POSIX_THREAD_SPORADIC_SERVER} is defined, the *sched_param* structure defined in `<sched.h>` shall contain the following members in addition to those specified above:

Member Type	Member Name	Description
int	*sched_ss_low_priority*	Low scheduling priority for sporadic server.
timespec	*sched_ss_repl_period*	Replenishment period for sporadic server.
timespec	*sched_ss_init_budget*	Initial budget for sporadic server.
int	*sched_ss_max_repl*	Maximum pending replenishments for sporadic server.

13.2 Scheduling Policies

⇒ **13.2 Scheduling Policies** *Add the following after the unnumbered table with the scheduling policies that shall be defined in* `<sched.h>`*:*

If {_POSIX_SPORADIC_SERVER} or {_POSIX_THREAD_SPORADIC_SERVER} is defined, then the following scheduling policy is provided in `<sched.h>`:

Symbol	Description
SCHED_SPORADIC	Sporadic server scheduling policy.

13.2.3 SCHED_OTHER

⇒ **13.2.3 SCHED_OTHER** *Change the sentence beginning with* "The effect of scheduling threads with the . . . " *to the following:*

The effect of scheduling threads with the SCHED_OTHER policy in a system in which other threads are executing under SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC shall thus be implementation defined.

⇒ **13.2 Scheduling Policies** *Add the following subclause:*

13.2.4 SCHED_SPORADIC

If {_POSIX_SPORADIC_SERVER} is defined or {_POSIX_THREAD_SPORADIC_-SERVER} is defined, the implementation shall include a scheduling policy identified by the value SCHED_SPORADIC.

The sporadic server policy is based primarily on two parameters: the replenishment period and the available execution capacity. The replenishment period is given by the *sched_ss_repl_period* member of the *sched_param* structure. The available execution capacity is initialized to the value given by the *sched_ss_init_budget* member of the same parameter. The sporadic server policy is identical to the SCHED_FIFO policy with some additional conditions that cause the thread's assigned priority to be switched between the values specified by the *sched_priority* and *sched_ss_low_priority* members of the *sched_param* structure.

The priority assigned to a thread using the sporadic server scheduling policy is determined in the following manner: If the available execution capacity is greater than zero and the number of pending replenishment operations is strictly less than *sched_ss_max_repl*, the thread is assigned the priority specified by *sched_priority*. Otherwise, the assigned priority shall be *sched_ss_low_priority*. If the value of *sched_priority* is less than or equal to the value of *sched_ss_low_priority*, the results are undefined. When active, the thread shall belong to the thread list corresponding to its assigned priority level, according to the mentioned priority assignment. The modification of the available execution capacity and, consequently of the assigned priority, is done as follows:

(1) When the thread at the head of the *sched_priority* list becomes a running thread, its execution time shall be limited to at most its available execution capacity, plus the resolution of the execution time clock used for this scheduling policy. This resolution shall be implementation defined.

(2) Each time the thread is inserted at the tail of the list associated with *sched_priority* (because as a blocked thread it became runnable with priority *sched_priority* or because a replenishment operation was performed), the time at which this operation is done is posted as the *activation_time*.

(3) When the running thread with assigned priority equal to *sched_priority* becomes a preempted thread, it becomes the head of the thread list for its priority; and the execution time consumed is subtracted from the available execution capacity. If the available execution capacity would become negative by this operation, it shall be set to zero.

(4) When the running thread with assigned priority equal to *sched_priority* becomes a blocked thread, the execution time consumed is subtracted from the available execution capacity; and a replenishment operation is scheduled, as described in (6) and (7). If the available execution capacity would become negative by this operation, it shall be set to zero.

(5) When the running thread with assigned priority equal to *sched_priority* reaches the limit imposed on its execution time, it becomes the tail of the

thread list for *sched_ss_low_priority*; the execution time consumed is subtracted from the available execution capacity (which becomes zero); and a replenishment operation is scheduled, as described in (6) and (7).

(6) Each time a replenishment operation is scheduled, the amount of execution capacity to be replenished, *replenish_amount*, is set equal to the execution time consumed by the thread since the *activation_time*. The replenishment is scheduled to occur at *activation_time* plus *sched_ss_repl_period*. If the scheduled time obtained is before the current time, the replenishment operation is carried out immediately. Several replenishment operations may be pending at the same time, each of which will be serviced at its respective scheduled time. With the above rules, the number of replenishment operations simultaneously pending for a given thread that is scheduled under the sporadic server policy shall not be greater than *sched_ss_max_repl*.

(7) A replenishment operation consists of adding the corresponding *replenish_amount* to the available execution capacity at the scheduled time. If, as a consequence of this operation, the execution capacity would become larger than *sched_ss_initial_budget*, it shall be rounded down to a value equal to *sched_ss_initial_budget*. Additionally, if the thread was runnable or running and had an assigned priority equal to *sched_ss_low_priority*, then it becomes the tail of the thread list for *sched_priority*.

Execution time is defined in 2.2.2.

For this policy, changing the value of a CPU-time clock via *clock_settime*() shall have no effect on its behavior.

For this policy, valid priorities shall be within the range returned by the functions *sched_get_priority_min*() and *sched_get_priority_max*() when SCHED_SPORADIC is provided as the parameter. Conforming implementations shall provide a priority range of at least 32 distinct priorities for this policy.

13.3 Process Scheduling Functions

13.3.1 Set Scheduling Parameters

13.3.1.2 Description

⇒ **13.3.1.2 Set Scheduling Parameters—Description** *Add the following paragraphs to the description:*

If {_POSIX_SPORADIC_SERVER} is defined:

If the scheduling policy of the target process is SCHED_SPORADIC, the value specified by the *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget* members of the *param* argument shall represent the time parameters to be used by the sporadic server scheduling policy for the target process. The *sched_ss_max_repl* member of the *param* argument shall represent the maximum number of replenishments that are allowed to be pending simultaneously for the process scheduled under this scheduling policy.

The specified *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

The value of *sched_ss_max_repl* shall be within the inclusive range [1, {SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

If the scheduling policy of the target process is either SCHED_FIFO or SCHED_RR, the *sched_ss_low_priority*, *sched_ss_repl_period* and *sched_ss_init_budget* members of the *param* argument shall have no effect on the scheduling behavior. If the scheduling policy of this process is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, including SCHED_OTHER, the effects of these members shall be implementation defined.

⇒ **13.3.1.2 Set Scheduling Parameters—Description** *Replace the eighth paragraph, beginning* "If the current scheduling policy...," *with the following new paragraph:*

If the current scheduling policy for the process specified by *pid* is not SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the result is implementation defined; this case includes the SCHED_OTHER policy.

13.3.3 Set Scheduling Policy and Scheduling Parameters

13.3.3.2 Description

⇒ **13.3.3.2 Set Scheduling Policy and Scheduling Parameters—Description** *Add the following paragraphs to the description, before the last paragraph:*

If {_POSIX_SPORADIC_SERVER} is defined:

If the scheduling policy specified by *policy* is SCHED_SPORADIC, the value specified by the *sched_ss_low_priority* member of the *param* argument shall be any integer within the inclusive priority range for the sporadic server policy. The *sched_ss_repl_period* and *sched_ss_init_budget* members of the *param* argument shall represent the time parameters used by the sporadic server scheduling policy for the target process. The *sched_ss_max_repl* member of the *param* argument shall represent the maximum number of replenishments that are allowed to be pending simultaneously for the process scheduled under this scheduling policy.

The specified *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail.

The value of *sched_ss_max_repl* shall be within the inclusive range [1, {SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

If the scheduling policy specified by *policy* is either SCHED_FIFO or SCHED_RR, the *sched_ss_low_priority*, *sched_ss_repl_period*, and *sched_ss_init_budget* members of the *param* argument shall have no effect on the scheduling behavior.

13.4 Thread Scheduling

13.4.1 Thread Scheduling Attributes

⇒ **13.4.1 Thread Scheduling Attributes** *Add the following paragraph after the paragraph that begins with* "If the {_POSIX_THREAD_PRIORITY_SCHEDULING} option is defined, ...":

If {_POSIX_THREAD_SPORADIC_SERVER} is defined, the `schedparam` attribute supports four new members that are used for the sporadic server scheduling policy. These members are *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*. The meaning of these attributes is the same as in the definitions in 13.1.

13.4.3 Scheduling Allocation Domain

⇒ **13.4.3 Scheduling Allocation Domain** *Change the first sentence of the fourth paragraph, currently reading* "For application threads whose scheduling allocation domain size is greater than one, the rules defined for SCHED_FIFO and SCHED_RR in 13.2 shall be used in an implementation-defined manner." *to the following:*

For application threads whose scheduling allocation domain size is greater than one, the rules defined for SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC in 13.2 shall be used in an implementation-defined manner.

⇒ 13.4.3 Scheduling Allocation Domain

Add the following paragraph after the fourth paragraph in 13.4.3:

> If {_POSIX_THREAD_SPORADIC_SERVER} is defined, the rules defined for SCHED_SPORADIC in 13.2 shall be used in an implementation-defined manner for application threads whose scheduling allocation domain size is greater than one.

13.4.4 Scheduling Documentation

⇒ **13.4.4 Scheduling Documentation** *Change the first sentence, beginning with* "If {_POSIX_PRIORITY_SCHEDULING} is defined, then . . . " *and ending with* ". . . such a policy, are implementation defined." *to the following:*

If {_POSIX_PRIORITY_SCHEDULING} is defined, then any scheduling policy beyond SCHED_OTHER, SCHED_FIFO, SCHED_RR, and SCHED_SPORADIC, as well as the effects of the scheduling policies indicated by these other values, and the attributes required to support such a policy are implementation defined.

13.5 Thread Scheduling Functions

13.5.1 Thread Creation Scheduling Attributes

13.5.1.2 Description

⇒ **13.5.1.2 Thread Creation Scheduling Attributes—Description** *Add the following sentence to the sixth paragraph, beginning* "The suported values of *policy* ...":

> In addition, if {_POSIX_THREAD_SPORADIC_SERVER} is defined, the value of *policy* may be SCHED_SPORADIC.

Also, add the following sentences at the end of the eighth paragraph, which describes the functions pthread_attr_setschedparam*()* *and* pthread_attr_getschedparam*():*

> For the SCHED_SPORADIC policy, the required members of the *param* structure are *sched_priority*, *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*. The specified *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall be within the inclusive range [1, {SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

13.5.2 Dynamic Thread Scheduling Parameters Access

13.5.2.2 Description

⇒ **13.5.2.2 Dynamic Thread Scheduling Parameters Access—Description**
Add the following paragraph to the description, before the last paragraph:

If {_POSIX_THREAD_SPORADIC_SERVER} is defined:

> The *policy* argument may have the value SCHED_SPORADIC, with the exception for the *pthread_setschedparam*() function that, if the scheduling policy was not SCHED_SPORADIC at the time of the call, it is implementation defined whether the function is supported. In other words, the implementation need not allow the application to dynamically change the scheduling policy to SCHED_SPORADIC. The sporadic server scheduling policy has the associated parameters *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*, *sched_priority*, and *sched_ss_max_repl*. The specified *sched_ss_repl_period* shall be greater than or equal to the specified *sched_ss_init_budget* for the function to succeed; if it is not, then the function shall fail. The value of *sched_ss_max_repl* shall be within the inclusive range [1, {SS_REPL_MAX}] for the function to succeed; if not, the function shall fail.

13.5.2.4 Errors

⇒ **13.5.2.4 Dynamic Thread Scheduling Parameters Access—Errors** *Add the following error status value in the* "if detected" *section of the* pthread_setschedparam*() function:*

[ENOTSUP] An attempt was made to dynamically change the scheduling policy to SCHED_SPORADIC, and the implementation does not support this change.

Section 14: Clocks and Timers

14.2 Clock and Timer Functions

14.2.1 Clocks

14.2.1.2 Description

⇒ **14.2.1.2 Clock and Timer Functions—Description** *Add the following paragraphs to the description, after the paragraph starting "*A clock may be systemwide...*":*

If {_POSIX_CPUTIME} is defined, implementations shall support clock ID values [obtained by invoking *clock_getcpuclockid*()], which represent the CPU-time clock of a given process. Implementations shall also support the special *clockid_t* value CLOCK_PROCESS_CPUTIME_ID, which represents the CPU-time clock of the calling process when invoking one of the clock or timer functions. For these clock IDs, the values returned by *clock_gettime*() and specified by *clock_settime*() represent the amount of execution time of the process associated with the clock. Changing the value of a CPU-time clock via *clock_settime*() shall have no effect on the behavior of the sporadic server scheduling policy (see 13.2.4).

If {_POSIX_THREAD_CPUTIME} is defined, implementations shall support clock ID values [obtained by invoking *pthread_getcpuclockid*()], which represent the CPU-time clock of a given thread. Implementations shall also support the special *clockid_t* value CLOCK_THREAD_CPUTIME_ID, which represents the CPU-time clock of the calling thread when invoking one of the clock or timer functions. For these clock IDs, the values returned by *clock_gettime*() and specified by *clock_settime*() represent the amount of execution time of the thread associated with the clock. Changing the value of a CPU-time clock via *clock_settime*() shall have no effect on the behavior of the sporadic server scheduling policy (see 13.2.4).

14.2.2 Create a Per-Process Timer

14.2.2.2 Description

⇒ **14.2.2.2 Create a Per-Process Timer—Description** *Add the following paragraphs to the description, after the paragraph starting* "Each implementation shall define...":

If {_POSIX_CPUTIME} is defined, implementations shall support *clock_id* values representing the CPU-time clock of the calling process.

If {_POSIX_THREAD_CPUTIME} is defined, implementations shall support *clock_id* values representing the CPU-time clock of the calling thread.

It is implementation defined whether a *timer_create*() call will succeed if the value defined by *clock_id* corresponds to the CPU-time clock of a process or thread different from the process or thread invoking the function.

14.2.2.4 Errors

⇒ **14.2.2.4 Create a Per-Process Timer—Errors** *Add the following error condition:*

[ENOTSUP]
The implementation does not support the creation of a timer attached to the CPU-time clock that is specified by *clock_id* and associated with a process or thread different from the process or thread invoking *timer_create*().

⇒ **14 Clocks and Timers** *Add the following subclauses:*

14.3 Execution Time Monitoring

This subclause describes extensions to system interfaces to support monitoring and limitation of the execution time of processes and threads.

14.3.1 CPU-time Clock Characteristics

If {_POSIX_CPUTIME} is defined, process CPU-time clocks shall be supported in addition to the clocks described in 14.1.4.

If {_POSIX_THREAD_CPUTIME} is defined, thread CPU-time clocks shall be supported.

CPU-time clocks measure execution or CPU time, which is defined in 2.2.2. The mechanism used to measure execution time is described in 2.3.1.

If {_POSIX_CPUTIME} is defined, the following constant of the type *clockid_t* shall be defined in `<time.h>`:

`CLOCK_PROCESS_CPUTIME_ID`
: When this value of the type *clockid_t* is used in a clock or timer function call, it is interpreted as the identifier of the CPU-time clock associated with the process making the function call.

If {_POSIX_THREAD_CPUTIME} is defined, the following constant of the type *clockid_t* shall be defined in `<time.h>`:

`CLOCK_THREAD_CPUTIME_ID`
: When this value of the type *clockid_t* is used in a clock or timer function call, it is interpreted as the identifier of the CPU-time clock associated with the thread making the function call.

14.3.2 Accessing a Process CPU-time Clock

Function: *clock_getcpuclockid*().

14.3.2.1 Synopsis

```
#include <sys/types.h>
#include <time.h>
int clock_getcpuclockid (pid_t pid, clockid_t *clock_id);
```

14.3.2.2 Description

If {_POSIX_CPUTIME} is defined:

> The *clock_getcpuclockid*() function shall return the clock ID of the CPU-time clock of the process specified by *pid*. If the process described by *pid* exists and the calling process has permission, the clock ID of this clock shall be returned in *clock_id*.
>
> If *pid* is zero, the *clock_getcpuclockid*() function shall return in *clock_id* the clock ID of the CPU-time clock of the process making the call.
>
> The conditions under which one process has permission to obtain the CPU-time clock ID of other processes are implementation defined.

Otherwise:

> Either the implementation shall support the *clock_getcpuclockid*() function as described above, or this function shall not be provided.

14.3.2.3 Returns

Upon successful completion, *clock_getcpuclockid*() shall return zero. Otherwise, the corresponding error value shall be returned.

14.3.2.4 Errors

If the following conditions occur, the *clock_getcpuclockid*() function shall return the corresponding error number:

[EPERM]
: The requesting process does not have permission to access the CPU-time clock for the process.

If the following condition is detected, the *clock_getcpuclockid*() function shall return the corresponding error number:

[ESRCH]
: No process can be found corresponding to the value specified by *pid*.

14.3.2.5 Cross-References

clock_gettime(), 14.2.1; *clock_settime*(), 14.2.1; *clock_getres*(), 14.2.1; *timer_create*(), 14.2.2.

14.3.3 Accessing a Thread CPU-time Clock

Function: *pthread_getcpuclockid*().

14.3.3.1 Synopsis

```
#include <sys/types.h>
#include <time.h>
#include <pthread.h>

int pthread_getcpuclockid (pthread_t thread_id, clockid_t *clock_id);
```

14.3.3.2 Description

If {_POSIX_THREAD_CPUTIME} is defined:

> The *pthread_getcpuclockid*() function shall return in *clock_id* the clock ID of the CPU-time clock of the thread specified by *thread_id*, if the thread specified by *thread_id* exists.

Otherwise:

> Either the implementation shall support the *pthread_getcpuclockid*() function as described above, or this function shall not be provided.

14.3.3.3 Returns

Upon successful completion, *pthread_getcpuclockid*() shall return zero. Otherwise the corresponding error number shall be returned.

14.3.3.4 Errors

If the following condition is detected, the *pthread_getcpuclockid*() function shall return the corresponding error number:

[ESRCH]
The value specified by *thread_id* does not refer to an existing thread.

14.3.3.5 Cross-References

clock_gettime(), 14.2.1; *clock_settime*(), 14.2.1; *clock_getres*(), 14.2.1; *clock_getcpuclockid*(), 14.3.2; *timer_create*(), 14.2.2;

Section 15: Message Passing

15.2 Message Passing Functions

15.2.4 Send a Message to a Message Queue

⇒ **15.2.4 Send a Message to a Message Queue** *Add the following function at the end of the list and change* "Function" *to* "Functions"*:*

mq_timedsend()

15.2.4.1 Synopsis

⇒ **15.2.4.1 Send a Message to a Message Queue—Synopsis**
Add the following #include and prototype to the end of the synopsis:

```
#include <time.h>
int mq_timedsend(mqd_t mqdes,
                 const char *msg_ptr,
                 size_t msg_len,
                 unsigned int msg_prio,
                 const struct timespec *abs_timeout);
```

15.2.4.2 Description

⇒ **15.2.4.2 Send a Message to a Message Queue—Description** *Add the following text at the end of the description:*

If {_POSIX_MESSAGE_PASSING} and {_POSIX_TIMEOUTS} are both defined:

> The *mq_timedsend*() function adds a message to the message queue specified by *mqdes* in the manner defined for the *mq_send*() function. However, if the specified message queue is full and O_NONBLOCK is not set in the message queue description associated with *mqdes*, the wait for sufficient room in the queue shall be terminated when the specified timeout expires. If O_NONBLOCK is set in the message queue description, this function shall behave identically to *mq_send*().
>
> The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (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. If the Timers option is supported, the timeout is based on the CLOCK_REALTIME clock. If the Timers option is not supported, the timeout is based on the system clock as returned by the *time*() function. The resolution of the timeout is the resolution of the clock on which it is based. The *timespec* argument is defined as a structure in the header `<time.h>`.

Under no circumstance shall the operation fail with a timeout if there is sufficient room in the queue to add the message immediately. The validity of the *abs_timeout* parameter need not be checked when there is sufficient room in the queue.

Otherwise:

Either the implementation shall support the *mq_timedsend*() function as described above, or this function shall not be provided.

15.2.4.3 Returns

⇒ **15.2.4.3 Send a Message to a Message Queue—Returns** *Add the following function at the end of the list and change* "Function" *to* "Functions"*:*

mq_timedsend()

15.2.4.4 Errors

⇒ **15.2.4.4 Send a Message to a Message Queue—Errors** *Make the following changes to the discussion of error conditions:*

Add *mq_timedsend*() at the end of the list of functions to which the error conditions apply.

Add an [ETIMEDOUT] error value (in alphabetical order) with the following reason:

The O_NONBLOCK flag was not set when the message queue was opened, but the timeout expired before the message could be added to the queue.

To the [EINVAL] error description, add the following reason:

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

Add *mq_timedsend*() to the list of functions returning [EINTR].

15.2.4.5 Cross-References

⇒ **15.2.4.5 Send a Message to a Message Queue—Cross-References**

Add the following cross references to the list, in alphabetical order:

mq_open(), 15.2.1; *time*() 4.5.1; `<time.h>`, 14.1.

15.2.5 Receive a Message from a Message Queue

⇒ **15.2.5 Receive a Message from a Message Queue** *Add the following function at the end of the list and change* "Function" *to* "Functions"*:*

mq_timedreceive()

15.2.5.1 Synopsis

⇒ **15.2.5.1 Receive a Message from a Message Queue—Synopsis**
Add the following #include and prototype to the end of the synopsis:

```
#include <time.h>
int mq_timedreceive(mqd_t mqdes,
                    char *msg_ptr,
                    size_t msg_len,
                    unsigned int *msg_prio,
                    const struct timespec *abs_timeout);
```

15.2.5.2 Description

⇒ **15.2.5.2 Receive a Message from a Message Queue—Description** *Add the following text at the end of the description:*

If {_POSIX_MESSAGE_PASSING} and {_POSIX_TIMEOUTS} are both defined:

> The *mq_timedreceive*() function is used to receive the oldest of the highest priority messages from the message queue specified by *mqdes* as in the *mq_receive*() function. However, if O_NONBLOCK was not specified when the message queue was opened via the *mq_open*() function and no message exists on the queue to satisfy the receive, the wait for such a message will be terminated when the specified timeout expires. If O_NONBLOCK is set, this function shall behave identically to *mq_receive*().
>
> The timeout expires when the absolute time specified by *abs_timeout* passes, as measured by the clock on which timeouts are based (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. If the Timers option is supported, the timeout is based

on the CLOCK_REALTIME clock; if the Timers option is not supported, the timeout is based on the system clock as returned by the *time*() function. The resolution of the timeout is the resolution of the clock on which it is based. The *timespec* argument is defined as a structure in the header `<time.h>`.

Under no circumstance shall the operation fail with a timeout if a message can be removed from the message queue immediately. The validity of the *abs_timeout* parameter need not be checked if a message can be removed from the message queue immediately.

Otherwise:

Either the implementation shall support the *mq_timedreceive*() function as described above, or this function shall not be provided.

15.2.5.3 Returns

⇒ **15.2.5.3 Receive a Message from a Message Queue—Returns** *Add the following function to the list of functions:*

mq_timedreceive()

15.2.5.4 Errors

⇒ **15.2.5.4 Receive a Message from a Message Queue—Errors** *Make the following changes to the discussion of error conditions:*

Add *mq_timedreceive*() at the end of the list of functions for both the "if occurs" error conditions and the "if detected" error conditions.

Add an [ETIMEDOUT] error value to the "if occurs" error conditions (in alphabetical order), with the following reason:

The O_NONBLOCK flag was not set when the message queue was opened, but no message arrived on the queue before the specified timeout expired.

Add an [EINVAL] error value to the "if occurs" error conditions (in alphabetical order), with the following reason:

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

Add *mq_timedreceive*() to the list of functions returning [EINTR].

15.2.5.5 Cross-References

⇒ 15.2.5.5 Receive a Message from a Message Queue—Cross-References

Add the following cross-references in alphabetical order:

mq_open(), 15.2.1; *time*(), 4.5.1; `<time.h>`, 14.1.

Section 16: Thread Management

16.2 Thread Functions

16.2.2 Thread Creation

16.2.2.2 Description

⇒ **16.2.2.2 Thread Creation—Description** *Add the following paragraph to the description, after the paragraph starting* "The signal state of the new thread..."*:*

> If {_POSIX_THREAD_CPUTIME} is defined, the new thread shall have a CPU-time clock accessible, and the initial value of this clock shall be set to zero.

Section 18: Thread Cancellation

18.1 Thread Cancellation Overview

18.1.2 Cancellation Points

⇒ **18.1.2 Cancellation Points** *Add the following functions (in alphabetical order) to the list of functions for which a cancellation point shall occur:*

mq_timedsend(), *mq_timedreceive*(), *sem_timedwait*().

⇒ **18.1.2 Cancellation Points** *Add the following functions (in alphabetical order) to the list of functions for which a cancellation point may also occur:*

posix_fadvise(), *posix_fallocate*(), *posix_madvise*(), *posix_spawn*(), *posix_spawnp*().

Section 19: Advisory Information

⇒ **19 Advisory Information** *Add the following section:*

19.1 I/O Advisory Information and Space Control

19.1.1 File Advisory Information

Function: *posix_fadvise*().

19.1.1.1 Synopsis

```
#include <sys/types.h>
#include <fcntl.h>

int posix_fadvise(int fd, off_t offset,
                  size_t len, int advice);
```

19.1.1.2 Description

If {_POSIX_ADVISORY_INFO} is defined:

The *posix_fadvise*() function provides advice to the implementation on the expected behavior of the application with respect to the data in the file associated with the open file descriptor, *fd*, starting at *offset* and continuing for *len* bytes. The specified range need not currently exist in the file. If *len* is zero, all data following *offset* is specified. The implementation may use this information to optimize handling of the specified data. The *posix_fadvise*() function has no effect on the semantics of other operations on the specified data although it may affect the performance of other operations.

The advice to be applied to the data is specified by the *advice* parameter and may be one of the following values:

`POSIX_FADV_NORMAL` specifies that the application has no advice to give on its behavior with respect to the specified data. It is the default characteristic if no advice is given for an open file.

`POSIX_FADV_SEQUENTIAL` specifies that the application expects to access the specified data sequentially from lower offsets to higher offsets.

`POSIX_FADV_RANDOM` specifies that the application expects to access the specified data in a random order.

`POSIX_FADV_WILLNEED` specifies that the application expects to access the specified data in the near future.

`POSIX_FADV_DONTNEED` specifies that the application expects that it will not access the specified data in the near future.

`POSIX_FADV_NOREUSE` specifies that the application expects to access the specified data once and then not reuse them thereafter.

These values shall be defined in `<fcntl.h>` if the Advisory Information option is supported.

Otherwise:

Either the implementation shall support the *posix_fadvise*() function as described above, or this function shall not be provided.

19.1.1.3 Returns

Upon successful completion, the *posix_fadvise*() function shall return a value of zero; otherwise, it shall return an error number to indicate the error.

19.1.1.4 Errors

If any of the following conditions occur, the *posix_fadvise*() function shall return the corresponding error number:

[EBADF] The *fd* argument is not a valid file descriptor.

[ESPIPE] The *fd* argument is associated with a pipe or FIFO.

[EINVAL] The value in *advice* is invalid.

19.1.1.5 Cross-References

posix_madvise(), 19.2.1.

19.1.2 File Space Control

Function: *posix_fallocate*().

19.1.2.1 Synopsis

```
#include <sys/types.h>
#include <fcntl.h>

int posix_fallocate(int fd, off_t offset, size_t len);
```

19.1.2.2 Description

If {_POSIX_ADVISORY_INFO} is defined:

> The *posix_fallocate*() function ensures that any required storage for regular file data starting at *offset* and continuing for *len* bytes is allocated on the file system storage media. If *posix_fallocate*() returns successfully, subsequent writes to the specified file data shall not fail due to the lack of free space on the file system storage media.
>
> If the *offset* + *len* is beyond the current file size, then *posix_fallocate*() shall adjust the file size to *offset* + *len*. Otherwise, the file size shall not be changed.
>
> It is implementation defined whether a previous *posix_fadvise*() call influences allocation strategy.
>
> Space allocated via *posix_fallocate*() shall be freed by a successful call to *creat*() or *open*() that truncates the size of the file. Space allocated via *posix_fallocate*() may be freed by a successful call to *ftruncate*() that reduces the file size to a size smaller than *offset* + *len*.

Otherwise:

> Either the implementation shall support the *posix_fallocate*() function as described above, or this function shall not be provided.

19.1.2.3 Returns

Upon successful completion, the *posix_fallocate*() function shall return a value of zero; otherwise, it shall return an error number to indicate the error.

19.1.2.4 Errors

If any of the following conditions occur, the *posix_fallocate*() function shall return the corresponding error number:

[EBADF] The *fd* argument is not a valid file descriptor.

[EBADF] The *fd* argument references a file that was opened without write permission.

[EFBIG] The value of *offset* + *len* is greater than the maximum file size.

[EINTR] A signal was caught during execution.

[EINVAL] The *len* argument was zero or the *offset* argument was less than zero.

[EIO] An I/O error occurred while reading from or writing to a file system.

[ENODEV] The *fd* argument does not refer to a regular file.

[ENOSPC] There is insufficient free space remaining on the file system storage media.

[ESPIPE] The *fd* argument is associated with a pipe or FIFO.

19.1.2.5 Cross-References

unlink(), 5.5.1; *open*(), 5.3.1; *creat*(), 5.3.2; *ftruncate*(), 5.6.7.

19.2 Memory Advisory Information and Alignment Control

19.2.1 Memory Advisory Information

Function: *posix_madvise*().

19.2.1.1 Synopsis

```
#include <sys/types.h>
#include <sys/mman.h>

int posix_madvise(void *addr, size_t len, int advice);
```

19.2.1.2 Description

If {_POSIX_ADVISORY_INFO} is defined and either {_POSIX_MAPPED_FILES} or {_POSIX_SHARED_MEMORY_OBJECTS} is defined:

The *posix_madvise*() function provides advice to the implementation on the expected behavior of the application with respect to the data in the memory starting at address, *addr*, and continuing for *len* bytes. The implementation may use this information to optimize handling of the specified data. The *posix_madvise*() function has no effect on the semantics of access to memory in the specified range although it may affect the performance of access.

The implementation may require that *addr* be a multiple of the page size, which is the value returned by *sysconf*() when the *name* value _SC_PAGESIZE is used.

The advice to be applied to the memory range is specified by the *advice* parameter and may be one of the following values:

`POSIX_MADV_NORMAL` specifies that the application has no advice to give on its behavior with respect to the specified range. It is the default characteristic if no advice is given for a range of memory.

`POSIX_MADV_SEQUENTIAL` specifies that the application expects to access the specified range sequentially from lower addresses to higher addresses.

`POSIX_MADV_RANDOM` specifies that the application expects to access the specified range in a random order.

`POSIX_MADV_WILLNEED` specifies that the application expects to access the specified range in the near future.

`POSIX_MADV_DONTNEED` specifies that the application expects that it will not access the specified range in the near future.

These values shall be defined in `<sys/mman.h>` if the Advisory Information option is supported and either the Memory Mapped Files option or the Shared Memory Objects option is supported.

Otherwise:

Either the implementation shall support the *posix_madvise*() function as described above, or this function shall not be provided.

19.2.1.3 Returns

Upon successful completion, the *posix_madvise*() function shall return a value of zero; otherwise, it shall return an error number to indicate the error.

19.2.1.4 Errors

If any of the following conditions occur, the *posix_madvise*() function shall return the corresponding error number:

[EINVAL] The value in *advice* is invalid.

[ENOMEM] Addresses in the range starting at *addr* and continuing for *len* bytes are partly or completely outside the range allowed for the address space of the calling process.

If any of the following conditions are detected, the *posix_madvise*() function shall return the corresponding error number:

[EINVAL] The value of *addr* is not a multiple of the value returned by *sysconf*() when the *name* value _SC_PAGESIZE is used.

[EINVAL] The value of *len* is zero.

19.2.1.5 Cross-References

posix_fadvise(), 19.1.1; *mmap*(), 12.2.1; *sysconf*(), 4.8.1.

19.2.2 Aligned Memory Allocation

Function: *posix_memalign*().

19.2.2.1 Synopsis

```
#include <sys/types.h>
#include <stdlib.h>

int posix_memalign(void **memptr, size_t alignment,
                   size_t size);
```

19.2.2.2 Description

If {_POSIX_ADVISORY_INFO} is defined:

The *posix_memalign*() function allocates *size* bytes aligned on a boundary specified by *alignment* and returns a pointer to the allocated memory in *memptr*. The value of *alignment* shall be a multiple of *sizeof*(void *) that is also a power of two. Upon successful completion, the value pointed to by *memptr* shall be a multiple of *alignment*.

The C Standard *free*() function deallocates memory that has previously been allocated by *posix_memalign*().

Otherwise:

Either the implementation shall support the *posix_memalign*() function as described above, or this function shall not be provided.

19.2.2.3 Returns

Upon successful completion, the *posix_memalign*() function returns a value of zero. Otherwise the *posix_memalign*() function shall return an error number to indicate the error.

19.2.2.4 Errors

If any of the following conditions occur, the *posix_memalign*() function shall return the corresponding error number:

[EINVAL] The value of the *alignment* parameter is not a power of two multiple of *sizeof*(void *).

[ENOMEM] There is insufficient memory available with the requested alignment.

19.2.2.5 Cross-References

free(), 8.1; *malloc*(), 8.1.

Annex A
(informative)
Bibliography

A.2 Other Standards

⇒ **A.2 Other Standards** *Add the following to the end of subclause A.2, with an appropriate reference number:*

{B1} ISO/IEC 14519:1998, *POSIX Ada Language Interfaces—Binding for System Application Interfaces (API) including Realtime Extensions.*

A.3 Historical Documentation and Introductory Texts

⇒ **A.3 Historical Documentation and Introductory Texts** *Add the following to the end of subclause A.3, with an appropriate reference number:*

{B2} Sprunt, B., Sha, L., and Lehoczky, J.P., "Aperiodic Task Scheduling for Hard Real-Time Systems." *The Journal of Real-Time Systems,* vol. 1, pp. 27-60, 1989.

Annex B
(informative)
Rationale and Notes

B.2 Definitions and General Requirements

B.2.3 General Concepts

⇒ **B.2.3 General Concepts:** *Add the following subclause, in the proper order, to the existing items in B.2.3:*

B.2.3.1 measurement of execution time

The methods used to measure the execution time of processes and threads, and the precision of these measurements, may vary considerably depending on the software architecture of the implementation and on the underlying hardware. Implementations can also make tradeoffs between the scheduling overhead and the precision of the execution time measurements. The standard does not impose any requirement on the accuracy of the execution time; it instead specifies that the measurement mechanism and its precision are implementation defined.

B.3 Process Primitives

B.3.1 Process Creation and Execution

⇒ **B.3.1 Process Creation and Execution** *Add the following subclauses:*

B.3.1.4 Spawn File Actions

A spawn file actions object may be initialized to contain an ordered sequence of close, dup2, and open operations to be used by *posix_spawn*() or *posix_spawnp*() to arrive at the set of open file descriptors inherited by the spawned process from the set of open file descriptors in the parent at the time of the *posix_spawn*() or *posix_spawnp*() call. It had been suggested that the close and dup2 operations alone are sufficient to rearrange file descriptors and that files which need be opened for use by the spawned process can be handled either by having the calling process open them before the *posix_spawn*() or *posix_spawnp*() call (and close them after) or by passing file names to the spawned process (in *argv*) so that it may open them itself. The working group recommends that applications use one of these two methods when practical since detailed error status on a failed open operation is always available to the application this way. However, the working group feels that allowing a spawn file actions object to specify open operations is still appropriate because

(1) It is consistent with equivalent POSIX.5 functionality (see the discussion on compatibility with POSIX.5 in B.3.1.6).

(2) It supports the I/O redirection paradigm commonly employed by POSIX programs designed to be invoked from a shell. When such a program is the child process, it may not be designed to open files on its own.

(3) It allows file opens that might otherwise fail or violate file ownership/access rights if executed by the parent process.

Regarding (2) above, the spawn open file action provides to *posix_spawn*() and *posix_spawnp*() the same capability that the shell redirection operators provide to *system*(), only without the intervening execution of a shell (e.g.: `system("myprog <file1 3<file2");`).

Regarding (3) above, if the calling process needs to open one or more files for access by the spawned process, but has insufficient spare file descriptors, then the open action is necessary to allow the open to occur in the context of the child process after other file descriptors (that must remain open in the parent) have been closed.

Additionally, if a parent is executed from a file having a ‘‘set-user-id’’ mode bit set and the POSIX_SPAWN_RESETIDS flag is set in the spawn attributes, a file created within the parent process will (possibly incorrectly) have the parent's effective user id as its owner whereas a file created via an open action during *posix_spawn*() or *posix_spawnp*() will have the parent's real id as its owner; and an open by the parent process may successfully open a file to which the real user should not have access or fail to open a file to which the real user should have access.

File Descriptor Mapping Rationale

The working group had originally proposed using an array that specified the mapping of child file descriptors back to the file descriptors of the parent. It was pointed out by the ballot group that it is not possible to reshuffle file descriptors arbitrarily in a library implementation of *posix_spawn*() or *posix_spawnp*() without provision for one or more spare file descriptor entries (which simply may not be available). Such an array requires that an implementation develop a complex strategy to achieve the desired mapping without inadvertently closing the wrong file descriptor at the wrong time.

It was noted by a member of the Ada Language Bindings working group that the approved Ada Language `Start_Process` family of POSIX process primitives uses a caller-specified set of file actions to alter the normal *fork*() / *exec* semantics for inheritance of file descriptors in a very flexible way, yet no such problems exist because the burden of determining how to achieve the final file descriptor mapping is completely on the application. Furthermore, although the file actions interface appears frightening at first glance, it is actually quite simple to implement in either a library or the kernel.

B.3.1.5 Spawn Attributes

The original spawn interface proposed in this standard defined the attributes that specify the inheritance of process attributes across a spawn operation as a structure. For the ability to separate optional individual attributes under their appropriate options (i.e., the `spawn-schedparam` and `spawn-schedpolicy` attributes depending upon the Process scheduling option) and also for extensibility and consistency with the newer POSIX interfaces, the attributes interface has been changed to an opaque datatype. This interface now consists of the type *posix_spawnattr_t*, representing a spawn attributes object, together with associated functions to initialize or destroy the attributes object, and to set or get each individual attribute. Although the new object-oriented interface is more verbose than the original structure, it is simple to use, more extensible, and easy to implement.

B.3.1.6 Spawn a Process

The POSIX *fork*() function is difficult or impossible to implement without swapping or dynamic address translation. POSIX needs process creation and file execution primitives that can be efficiently implemented without address translation or other MMU services, for the following reasons:

— Swapping is generally too slow for a realtime environment.

— Dynamic address translation is not available everywhere POSIX might be useful.

— Processes are too useful to simply option out of POSIX whenever it must run without address translation or other MMU services.

This function shall be called *posix_spawn*(). A closely related function, *posix_spawnp*(), is included for completeness.

The *posix_spawn*() function is implementable as a library routine, but both *posix_spawn*() and *posix_spawnp*() are designed as kernel operations. Also, although they may be an efficient replacement for many *fork*() / *exec* pairs, their goal is to provide useful process creation primitives for systems that have difficulty with *fork*(), not to provide drop-in replacements for *fork*() / *exec*.

This view of the role of *posix_spawn*() and *posix_spawnp*() influenced the design of their API. It does not attempt to provide the full functionality of *fork*() / *exec* in which arbitrary user-specified operations of any sort are permitted between the creation of the child process and the execution of the new process image; any attempt to reach that level would need to provide a programming language as parameters. Instead, *posix_spawn*() and *posix_spawnp*() are process creation primitives like the `Start_Process` and `Start_Process_Search` Ada language bindings in ISO/IEC 14519:1998 {B1} package `POSIX_Process_Primitives` and also like those in many operating systems that are not UNIX[1)] systems, but with some POSIX-specific additions.

To achieve their coverage goals, *posix_spawn*() and *posix_spawnp*() have control of six types of inheritance: file descriptors, process group ID, user and group ID, signal mask, scheduling, and whether each signal ignored in the parent will remain ignored in the child or be reset to its default action in the child.

Control of file descriptors is required to allow an independently written child process image to access data streams opened by and even generated or read by the parent process without being specifically coded to know which parent files and file descriptors are to be used. Control of the process group ID is required to control how the child process's job control relates to that of the parent.

Control of the signal mask and signal defaulting is sufficient to support the implementation of *system*() suggested in P1003.1a. Although support for *system*() is not explicitly one of the goals for *posix_spawn*() and *posix_spawnp*(), it is covered under the "at least 50%" coverage goal.

The intention is that the normal file descriptor inheritance across *fork*(), the subsequent effect of the specified spawn file actions, and the normal file descriptor inheritance across one of the *exec* family of functions should fully specify open file inheritance. The implementation need make no decisions regarding the set of open file descriptors when the child process image begins execution. Those decisions have already been made by the caller and expressed as the set of open file descriptors and their FD_CLOEXEC flags at the time of the call together with the spawn file actions object specified in the call. In the cases where the POSIX `Start_Process` Ada primitives have been implemented in a library, this method of controlling file descriptor inheritance may be implemented very easily. See Figure B-1 for a crude, but workable, C language implementation.

Several problems have been identified with *posix_spawn*() and *posix_spawnp*(), but a solution that introduces fewer problems does not appear to exist.

Environment modification for child process attributes not specifiable via the *attrp* or *file_actions* arguments shall be done in the parent process. Since the parent generally wants to save its context, it is more costly than similar functionality with *fork*() / *exec*. It is also complicated to modify the environment of a multithreaded process temporarily since all threads must agree when it is safe for the environment to be changed. However, this cost is only borne by those invocations of *posix_spawn*() and *posix_spawnp*() that use the additional functionality. Since extensive modifications are not the usual case and are particularly unlikely in time-critical code, keeping much of the environment control out of *posix_spawn*() and *posix_spawnp*() is appropriate design.

The *posix_spawn*() and *posix_spawnp*() functions do not have all the power of *fork*() / *exec*. The *fork*() function is a wonderfully powerful operation. Its functionality cannot be duplicated in a simple, fast function with no special hardware

---

1) UNIX is a registered trademark of The Open Group in the United States of America and other countries.

requirements. The *posix_spawn*() and *posix_spawnp*() functions are similar to the process creation operations on many operating systems that are not UNIX systems.

Requirements

The requirements for *posix_spawn*() and *posix_spawnp*() are as follows:

- They must be implementable without an MMU or unusual hardware.
- They must be compatible with existing POSIX standards.

Additional goals are the following:

- They should be efficiently implementable.
- They should be able to replace at least 50% of typical executions of *fork*().
- A system with *posix_spawn*() and *posix_spawnp*() and without *fork*() should be useful, at least for realtime applications.
- A system with *fork*() and the *exec* family should be able to implement *posix_spawn*() and *posix_spawnp*() as library routines.

Two-Syntax Rationale

POSIX *exec* has several calling sequences with approximately the same functionality. These appear to be required for compatibility with existing practice. Since the existing practice for the *posix_spawn* functions is otherwise substantially unlike POSIX, simplicity outweighs compatibility. There are, therefore, only two names for the *posix_spawn* functions.

The parameter list does not differ between *posix_spawn*() and *posix_spawnp*(); *posix_spawnp*() interprets the second parameter more elaborately than *posix_spawn*().

Compatibility with POSIX.5 `POSIX_Process_Primitives.Start_Process`

The `Start_Process` and `Start_Process_Search` procedures from ISO/IEC 14519:1998 {B1}, the Ada Language Binding to POSIX.1, encapsulate *fork*() and *exec* functionality in a manner similar to that of *posix_spawn*() and *posix_spawnp*(). Originally, in keeping with its simplicity goal, the working group had limited the capabilities of *posix_spawn*() and *posix_spawnp*() to a subset of the capabilities of `Start_Process` and `Start_Process_Search`; certain nondefault capabilities were not supported. However, based on suggestions by the ballot group to improve file descriptor mapping or drop it, and on the advice of an Ada Bindings working group member, the working group decided that *posix_spawn*() and *posix_spawnp*() should be sufficiently powerful to implement `Start_Process` and `Start_Process_Search`. The rationale is that if the Ada language binding to such a primitive had already been approved as an IEEE standard, there can be little justification for not approving the functionally equivalent parts of a C binding. The only three capabilities provided by *posix_spawn*() and *posix_spawnp*() that are not provided by `Start_Process` and `Start_Process_Search` are optionally specifying the child's process group id, the set of signals to be reset to default signal handling in the child process, and the child's scheduling policy and parameters.

For the Ada Language Binding for `Start_Process` to be implemented with *posix_spawn*(), that binding would need to explicitly pass an empty signal mask and the parent's environment to *posix_spawn*() whenever the caller of `Start_Process` allowed these arguments to default since *posix_spawn*() does not provide such defaults. The ability of `Start_Process` to mask user-specified signals during its execution is functionally unique to the Ada Language Binding and shall be dealt with in the binding separately from the call to *posix_spawn*().

Process Group

The process group inheritance field can be used to join the child process with an existing process group. By assigning a value of zero to the **`spawn-pgroup`** attribute of the object referenced by *attrp*, the *setpgid*() mechanism will place the child process in a new process group.

Threads

Without the *posix_spawn*() and *posix_spawnp*() functions, systems without address translation can still use threads to give an abstraction of concurrency. In many cases, thread creation suffices, but it is not always a good substitute. The *posix_spawn*() and *posix_spawnp*() functions are considerably "*heavier*" than thread creation. Processes have several important attributes that threads do not. Even without address translation, a process may have base-and-bound memory protection. Each process has a process environment including security attributes, file capabilities, and powerful scheduling attributes specified by POSIX.1 and POSIX.1b. Processes abstract the behavior of nonuniform-memory-architecture multiprocessors better than threads, and they are more convenient to use for activities that are not closely linked.

The *posix_spawn*() and *posix_spawnp*() functions may not bring support for multiple processes to every configuration. Process creation is not the only piece of operating system support required to support multiple processes. The total cost of support for multiple processes may be quite high in some circumstances. Existing practice shows that support for multiple processes is uncommon and threads are common among "*tiny kernels*." There should, therefore, probably continue to be AEPs for operating systems with only one process.

Asynchronous Error Notification Rationale

A library implementation of *posix_spawn*() or *posix_spawnp*() may not be able to detect all possible errors before it forks the child process. This standard provides for an error indication returned from a child process, which could not successfully complete the spawn operation, via a special exit status that may be detected using the status value returned by *wait*() and *waitpid*().

The *stat_val* interface and the macros used to interpret it are not well-suited to the purpose of returning API errors, but they are the only path available to a library implementation. Thus, an implementation may cause the child process to exit with exit status 127 for any error detected during the spawn process after the *posix_spawn*() or *posix_spawnp*() function has successfully returned.

The working group had proposed using two additional macros to interpret *stat_val*: First, WIFSPAWNFAIL would have detected a status that indicated that the child exited because of an error detected during the *posix_spawn*() or *posix_spawnp*() operations rather than during actual execution of the child process image. Second, WSPAWNERRNO would have extracted the error value if WIFSPAWNFAIL indicated a failure. The balloting group strongly opposed this approach because it would make a library implementation of *posix_spawn*() or *posix_spawnp*() dependent on kernel modifications to *waitpid*() to be able to embed special information in *stat_val* to indicate a spawn failure.

The 8 bits of child process exit status that are guaranteed by this standard to be accessible to the waiting parent process are insufficient to disambiguate a spawn error from any other kind of error that may be returned by an arbitrary process image. No other bits of the exit status are required to be visible in *stat_val*, so these macros could not be strictly implemented at the library level. Reserving an exit status of 127 for such spawn errors is consistent with the use of this value by *system*() and *popen*() to signal failures in these operations that occur after the function has returned, but before a shell is able to execute. The exit status of 127 does not uniquely identify this class of error, nor does it provide any detailed information on the nature of the failure. A kernel implementation of *posix_spawn*() or *posix_spawnp*() is permitted (and encouraged) to return any possible error as the function value, thus providing more detailed failure information to the parent process.

Thus, no special macros are available to isolate asynchronous *posix_spawn*() or *posix_spawnp*() errors. Instead, errors detected by the *posix_spawn*() or *posix_spawnp*() operations in the context of the child process before the new process image executes are reported by setting the child's exit status to 127. The calling process may use the WIFEXITED and WEXITSTATUS macros on the *stat_val* stored by the *wait*() or *waitpid*() functions to detect spawn failures to the extent that other status values with which the child process image may exit (before the parent can conclusively determine whether the child process image has begun execution) are distinct from exit status 127.

Library Implementation of Spawn

The *posix_spawn*() or *posix_spawnp*() operation is enough to

- Simply start a process executing a process image. This application is the simplest for process creation, and it may cover most executions of POSIX *fork*().
- Support I/O redirection, including pipes.
- Run the child under a user and group ID in the domain of the parent.
- Run the child at any priority in the domain of the parent.

The *posix_spawn*() or *posix_spawnp*() operation does not cover every possible use of *fork*(), but it does span the common applications: typical use by `shell` and `login`.

The cost is that before it calls *posix_spawn*() or *posix_spawnp*(), the parent must adjust to a state that *posix_spawn*() or *posix_spawnp*() can map to the desired state for the child. Environment changes require the parent to save some of its state and restore it afterwards. The effective behavior of a successful invocation of *posix_spawn*() is as if the operation were implemented with POSIX operations as shown in Figure B-1.

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sched.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <string.h>
#include <signal.h>

/*#include <spawn.h>*/
/*************************************/
/*Things that could be defined in spawn.h*/
/*************************************/
typedef struct
   {
   short posix_attr_flags;
#define POSIX_SPAWN_SETPGROUP       0x1
#define POSIX_SPAWN_SETSIGMASK      0x2
#define POSIX_SPAWN_SETSIGDEF       0x4
#define POSIX_SPAWN_SETSCHEDULER    0x8
#define POSIX_SPAWN_SETSCHEDPARAM   0x10
#define POSIX_SPAWN_RESETIDS        0x20
   pid_t posix_attr_pgroup;
   sigset_t posix_attr_sigmask;
   sigset_t posix_attr_sigdefault;
   int posix_attr_schedpolicy;
   struct sched_param posix_attr_schedparam;
   } posix_spawnattr_t;

typedef char *posix_spawn_file_actions_t;

int posix_spawn_file_actions_init(
          posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_destroy(
          posix_spawn_file_actions_t *file_actions);
int posix_spawn_file_actions_addclose(
          posix_spawn_file_actions_t *file_actions,
          int fildes);
int posix_spawn_file_actions_adddup2(
          posix_spawn_file_actions_t *file_actions,
          int fildes, int newfildes);
int posix_spawn_file_actions_addopen(
          posix_spawn_file_actions_t *file_actions,
          int fildes, const char *path, int oflag,
          mode_t mode);
int posix_spawnattr_init (
          posix_spawnattr_t *attr);
int posix_spawnattr_destroy (
          posix_spawnattr_t *attr);
int posix_spawnattr_getflags (
          const posix_spawnattr_t *attr,
          short *flags);
int posix_spawnattr_setflags (
```

```
          posix_spawnattr_t *attr,
          short flags);
int posix_spawnattr_getpgroup (
          const posix_spawnattr_t *attr,
          pid_t *pgroup);
int posix_spawnattr_setpgroup (
          posix_spawnattr_t *attr,
          pid_t pgroup);
int posix_spawnattr_getschedpolicy (
          const posix_spawnattr_t *attr,
          int *schedpolicy);
int posix_spawnattr_setschedpolicy (
          posix_spawnattr_t *attr,
          int schedpolicy);
int posix_spawnattr_getschedparam (
          const posix_spawnattr_t *attr,
          struct sched_param *schedparam);
int posix_spawnattr_setschedparam (
          posix_spawnattr_t *attr,
          const struct sched_param *schedparam);
int posix_spawnattr_getsigmask (
          const posix_spawnattr_t *attr,
          sigset_t *sigmask);
int posix_spawnattr_setsigmask (
          posix_spawnattr_t *attr,
          const sigset_t *sigmask);
int posix_spawnattr_getsigdefault (
          const posix_spawnattr_t *attr,
          sigset_t *sigdefault);
int posix_spawnattr_setsigdefault (
          posix_spawnattr_t *attr,
          const sigset_t *sigdefault);
int posix_spawn(
          pid_t *pid,
          const char *path,
          const posix_spawn_file_actions_t *file_actions,
          const posix_spawnattr_t *attrp,
          char * const argv[],
          char * const envp[]);
int posix_spawnp(
          pid_t *pid,
          const char *file,
          const posix_spawn_file_actions_t *file_actions,
          const posix_spawnattr_t *attrp,
          char * const argv[],
          char * const envp[]);

/***************************************/
/*Example posix_spawn() library routine*/
/***************************************/
int posix_spawn(pid_t *pid,
      const char *path,
      const posix_spawn_file_actions_t *file_actions,
      const posix_spawnattr_t *attrp,
      char * const argv[],
      char * const envp[])
```

```
    {
    /*Create process*/
    if((*pid=fork()) == (pid_t)0)
       {
       /*This is the child process*/
       /*Worry about process group*/
       if(attrp->posix_attr_flags & POSIX_SPAWN_SETPGROUP)
          {
          /*Override inherited process group*/
          if(setpgid(0, attrp->posix_attr_pgroup) != 0)
             {
             /*Failed*/
             exit(127);
             }
          }

       /*Worry about process signal mask*/
       if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGMASK)
          {
          /*Set the signal mask (can't fail)*/
          sigprocmask(SIG_SETMASK, &attrp->posix_attr_sigmask,
                 NULL);
          }

       /*Worry about resetting effective user and group IDs*/
       if(attrp->posix_attr_flags & POSIX_SPAWN_RESETIDS)
          {
          /*None of these can fail for this case.*/
          setuid(getuid());
          setgid(getgid());
          }

       /*Worry about defaulted signals*/
       if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGDEF)
          {
          struct sigaction deflt;
          sigset_t all_signals;
          int s;

          /*Construct default signal action*/
          deflt.sa_handler = SIG_DFL;
          deflt.sa_flags = 0;

          /*Construct the set of all signals*/
          sigfillset(&all_signals);

          /*Loop for all signals*/
          for(s=0; sigismember(&all_signals,s); s++)
             {
             /*Signal to be defaulted?*/
             if(sigismember(&attrp->posix_attr_sigdefault,s))
                {
                /*Yes - default this signal*/
                if(sigaction(s, &deflt, NULL) == -1)
                   {
                   /*Failed*/
```

```
            exit(127);
            }
         }
      }
   }

/*Worry about the fds if we are to map them*/
if(file_actions != NULL)
   {
   /*Loop for all actions in object *file_actions*/
   /*(implementation dives beneath abstraction)*/
   char *p = *file_actions;
   while(*p != '\0')
      {
      if(strncmp(p,"close(",6) == 0)
         {
         int fd;
         if(sscanf(p+6,"%d)",&fd) != 1)
            {
            exit(127);
            }
         if(close(fd) == -1) exit(127);
         }
      else if(strncmp(p,"dup2(",5) == 0)
         {
         int fd,newfd;
         if(sscanf(p+5,"%d,%d)",&fd,&newfd) != 2)
            {
            exit(127);
            }
         if(dup2(fd, newfd) == -1) exit(127);
         }
      else if(strncmp(p,"open(",5) == 0)
         {
         int fd,oflag;
         mode_t mode;
         int tempfd;
         char path[1000]; /*should be dynamic*/
         char *q;
         if(sscanf(p+5,"%d,",&fd) != 1)
            {
            exit(127);
            }
         p = strchr(p, ',') + 1;
         q = strchr(p, '*');
         if(q == NULL) exit(127);
         strncpy(path, p, q-p);
         path[q-p] = '\0';
         if(sscanf(q+1,"%o,%o)",&oflag,&mode)!=2)
            {
            exit(127);
            }
         if(close(fd) == -1)
            {
            if(errno != EBADF) exit(127);
            }
```

```
                    tempfd = open(path, oflag, mode);
                    if(tempfd == -1) exit(127);
                    if(tempfd != fd)
                        {
                        if(dup2(tempfd,fd) == -1)
                            {
                            exit(127);
                            }
                        if(close(tempfd) == -1)
                            {
                            exit(127);
                            }
                        }
                    }
                else
                    {
                    exit(127);
                    }
                p = strchr(p, ')') + 1;
                }
            }

        /*Worry about setting new scheduling policy and parameters*/
        if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDULER)
            {
            if(sched_setscheduler(0, attrp->posix_attr_schedpolicy,
                &attrp->posix_attr_schedparam) == -1)
                {
                exit(127);
                }
            }

        /*Worry about setting only new scheduling parameters*/
        if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDPARAM)
            {
            if(sched_setparam(0, &attrp->posix_attr_schedparam)==-1)
                {
                exit(127);
                }
            }

        /*Now execute the program at path*/
        /*Any fd that still has FD_CLOEXEC set will be closed*/
        execve(path, argv, envp);
        exit(127); /*exec failed*/
        }
    else
        {
        /*This is the parent (calling) process*/
        if((int)pid == -1) return errno;
        return 0;
        }
    }

/*****************************************************/
/* Here is a crude but effective implementation of the */
```

```
/* file action object operators which store actions as */
/* concatenated token separated strings.               */
/*****************************************************/
/*Create object with no actions.*/
int posix_spawn_file_actions_init(
          posix_spawn_file_actions_t *file_actions)
   {
   *file_actions = malloc(sizeof(char));
   if(*file_actions == NULL) return ENOMEM;
   strcpy(*file_actions, "");
   return 0;
   }

/*Free object storage and make invalid.*/
int posix_spawn_file_actions_destroy(
          posix_spawn_file_actions_t *file_actions)
   {
   free(*file_actions);
   *file_actions = NULL;
   return 0;
   }

/*Add a new action string to object.*/
static int add_to_file_actions(
            posix_spawn_file_actions_t *file_actions,
            char *new_action)
   {
   *file_actions = realloc
      (*file_actions, strlen(*file_actions)+strlen(new_action)+1);
   if(*file_actions == NULL) return ENOMEM;
   strcat(*file_actions, new_action);
   return 0;
   }

/*Add a close action to object.*/
int posix_spawn_file_actions_addclose(
          posix_spawn_file_actions_t *file_actions,
          int fildes)
   {
   char temp[100];
   sprintf(temp, "close(%d)", fildes);
   return add_to_file_actions(file_actions, temp);
   }

/*Add a dup2 action to object.*/
int posix_spawn_file_actions_adddup2(
          posix_spawn_file_actions_t *file_actions,
          int fildes, int newfildes)
   {
   char temp[100];
   sprintf(temp, "dup2(%d,%d)", fildes, newfildes);
   return add_to_file_actions(file_actions, temp);
   }

/*Add an open action to object.*/
int posix_spawn_file_actions_addopen(
```

```
            posix_spawn_file_actions_t *file_actions,
            int fildes, const char *path, int oflag,
            mode_t mode)
    {
    char temp[100];
    sprintf(temp, "open(%d,%s*%o,%o)", fildes, path, oflag, mode);
    return add_to_file_actions(file_actions, temp);
    }

/****************************************************/
/* Here is a crude but effective implementation of the  */
/* spawn attributes object functions which manipulate   */
/* the individual attributes.                            */
/****************************************************/
/*Initialize object with default values.*/
int posix_spawnattr_init (
            posix_spawnattr_t *attr)
    {
    attr->posix_attr_flags=0;
    attr->posix_attr_pgroup=0;
    /* Default value of signal mask is the parent's signal mask */
    /* other values are also allowed */
    sigprocmask(0,NULL,&attr->posix_attr_sigmask);
    sigemptyset(&attr->posix_attr_sigdefault);
    /* Default values of scheduling attr. inherited from the parent */
    /* other values are also allowed */
    attr->posix_attr_schedpolicy=sched_getscheduler(0);
    sched_getparam(0,&attr->posix_attr_schedparam);
    return 0;
    }

int posix_spawnattr_destroy (
            posix_spawnattr_t *attr)
    {
    /* No action needed */
    return 0;
    }

int posix_spawnattr_getflags (
            const posix_spawnattr_t *attr,
            short *flags)
    {
    *flags=attr->posix_attr_flags;
    return 0;
    }

int posix_spawnattr_setflags (
            posix_spawnattr_t *attr,
            short flags)
    {
    attr->posix_attr_flags=flags;
    return 0;
    }

int posix_spawnattr_getpgroup (
            const posix_spawnattr_t *attr,
```

```
          pid_t *pgroup)
   {
   *pgroup=attr->posix_attr_pgroup;
   return 0;
   }

int posix_spawnattr_setpgroup (
          posix_spawnattr_t *attr,
          pid_t pgroup)
   {
   attr->posix_attr_pgroup=pgroup;
   return 0;
   }

int posix_spawnattr_getschedpolicy (
          const posix_spawnattr_t *attr,
          int *schedpolicy)
   {
   *schedpolicy=attr->posix_attr_schedpolicy;
   return 0;
   }

int posix_spawnattr_setschedpolicy (
          posix_spawnattr_t *attr,
          int schedpolicy)
   {
   attr->posix_attr_schedpolicy=schedpolicy;
   return 0;
   }

int posix_spawnattr_getschedparam (
          const posix_spawnattr_t *attr,
          struct sched_param *schedparam)
   {
   *schedparam=attr->posix_attr_schedparam;
   return 0;
   }

int posix_spawnattr_setschedparam (
          posix_spawnattr_t *attr,
          const struct sched_param *schedparam)
   {
   attr->posix_attr_schedparam=*schedparam;
   return 0;
   }

int posix_spawnattr_getsigmask (
          const posix_spawnattr_t *attr,
          sigset_t *sigmask)
   {
   *sigmask=attr->posix_attr_sigmask;
   return 0;
   }

int posix_spawnattr_setsigmask (
          posix_spawnattr_t *attr,
```

```
        const sigset_t *sigmask)
    {
    attr->posix_attr_sigmask=*sigmask;
    return 0;
    }

int posix_spawnattr_getsigdefault (
        const posix_spawnattr_t *attr,
        sigset_t *sigdefault)
    {
    *sigdefault=attr->posix_attr_sigdefault;
    return 0;
    }

int posix_spawnattr_setsigdefault (
        posix_spawnattr_t *attr,
        const sigset_t *sigdefault)
    {
    attr->posix_attr_sigdefault=*sigdefault;
    return 0;
    }
```

Figure B-1 – *posix_spawn*() Equivalent

I/O redirection with *posix_spawn*() or *posix_spawnp*() is accomplished by crafting a *file_actions* argument to effect the desired redirection. Such a redirection follows the general outline of the example in Figure B-2.

```
/* To redirect new standard output (fd 1) to a file, */
/* and redirect new standard input (fd 0) from my fd socket_pair[1], */
/* and close my fd socket_pair[0] in the new process. */
posix_spawn_file_actions_t file_actions;
posix_spawn_file_actions_init    (&file_actions);
posix_spawn_file_actions_addopen (&file_actions, 1, "newout", ...);
posix_spawn_file_actions_dup2    (&file_actions, socket_pair[1], 0);
posix_spawn_file_actions_close   (&file_actions, socket_pair[0]);
posix_spawn_file_actions_close   (&file_actions, socket_pair[1]);
posix_spawn(..., &file_actions, ...)
posix_spawn_file_actions_destroy (&file_actions);
```

Figure B-2 – I/O Redirection with *posix_spawn*()

Spawning a process under a new *userid* uses the outline shown in Figure B-3.

```
Save = getuid();
setuid(newid);
posix_spawn(...)
setuid(Save);
```

Figure B-3 – Spawning a new Userid Process

B.13 Execution Scheduling

⇒ **B.13 Execution Scheduling** *Add the following subclause:*

B.13.3 Sporadic Server Scheduling Policy

The sporadic server is a mechanism defined for scheduling aperiodic activities in time-critical realtime systems. This mechanism reserves a certain bounded amount of execution capacity for processing aperiodic events at a high priority level. Any aperiodic events that cannot be processed within the bounded amount of execution capacity are executed in the background at a low priority level. Thus, a certain amount of execution capacity can be guaranteed to be available for processing periodic tasks, even under burst conditions in the arrival of aperiodic processing requests (i.e., a large number of requests in a short time interval). The sporadic server also simplifies the schedulability analysis of the realtime system because it allows aperiodic processes or threads to be treated as if they were periodic. The sporadic server was first described by Sprunt, et al. {B2}.

The key concept of the sporadic server is to provide and limit a certain amount of computation capacity for processing aperiodic events at their assigned normal priority, during a time interval called the replenishment period. Once the entity controlled by the sporadic server mechanism is initialized with its period and execution-time budget attributes, it preserves its execution capacity until an aperiodic request arrives. The request will be serviced (if no higher priority activities are pending) as long as execution capacity is left. If the request is completed, the actual execution time used to service it is subtracted from the capacity, and a replenishment of this amount of execution time is scheduled to happen one replenishment period after the arrival of the aperiodic request. If the request is not completed, because no execution capacity is left, then the aperiodic process or thread is assigned a lower background priority. For each portion of consumed execution capacity, the execution time used is replenished after one replenishment period. At the time of replenishment, if the sporadic server was executing at a background priority level, its priority is elevated to the normal level. Other similar replenishment policies have been defined, but the one presented here represents a compromise between efficiency and implementation complexity.

The interface that appears in this section defines a new scheduling policy for threads and processes that behaves according to the rules of the sporadic server mechanism. Scheduling attributes are defined and functions are provided to allow the user to set and get the parameters that control the scheduling behavior of this mechanism, namely the normal and low priority, the replenishment period, the maximum number of pending replenishment operations, and the initial execution-time budget.

B.13.3.1 Scheduling Aperiodic Activities (rationale)

Virtually all realtime applications are required to process aperiodic activities. In many cases, there are tight timing constraints that the response to the aperiodic events must meet. Usual timing requirements imposed on the response to these events are

— The effects of an aperiodic activity on the response time of lower priority activities must be controllable and predictable.

— The system must provide the fastest possible response time to aperiodic events.

— It must be possible to take advantage of all the available processing bandwidth not needed by time-critical activities to enhance average-case response times to aperiodic events.

Traditional methods for scheduling aperiodic activities are background processing, polling tasks, and direct event execution:

— Background processing consists of assigning a very low priority to the processing of aperiodic events. It utilizes all the available bandwidth in the system that has not been consumed by higher priority threads. However, it is difficult, or impossible, to meet requirements on average-case response time because the aperiodic entity has to wait for the execution of all other entities that have higher priority.

— Polling consists of creating a periodic process or thread for servicing aperiodic requests. At regular intervals, the polling entity is started, and it services accumulated pending aperiodic requests. If no aperiodic requests are pending, the polling entity suspends itself until its next period. Polling allows the aperiodic requests to be processed at a higher priority level. However, worst and average-case response times of polling entities are a direct function of the polling period, and there is execution overhead for each polling period, even if no event has arrived. If the deadline of the aperiodic activity is short compared to the interarrival time, the polling frequency must be increased to guarantee meeting the deadline. For this case, the increase in frequency can dramatically reduce the efficiency of the system and, therefore, its capacity to meet all deadlines. Yet, polling represents a good way to handle a large class of practical problems because it preserves system predictability and because the amortised overhead drops as load increases.

— Direct event execution consists of executing the aperiodic events at a high fixed-priority level. Typically, the aperiodic event is processed by an interrupt service routine as soon as it arrives. This technique provides predictable response times for aperiodic events, but makes the response times of all lower priority activities completely unpredictable under burst arrival conditions. Therefore, if the density of aperiodic event arrivals is unbounded, it may be a dangerous technique for time-critical systems. Yet, for cases in which the physics of the system imposes a bound on the event arrival rate, it is probably the most efficient technique.

The sporadic server scheduling algorithm combines the predictability of the polling approach with the short response times of the direct event execution. Thus, it allows systems to meet an important class of application requirements that cannot be met by using the traditional approaches. Multiple sporadic servers with different attributes can be applied to the scheduling of multiple classes of aperiodic events, each with different kinds of timing requirements, e.g., individual deadlines, average response times. It also has many other interesting applications for realtime, e.g., scheduling producer/consumer tasks in time-critical systems, limiting the effects of faults on the estimation of task execution-time requirements.

B.13.3.2 Existing Practice

The sporadic server has been used in different kinds of applications, e.g., military avionics, robot control systems, industrial automation systems. There are examples of many systems that cannot be successfully scheduled using the classic approaches, such as direct event execution or polling, and are schedulable using a sporadic server scheduler. The sporadic server algorithm itself can successfully schedule all systems scheduled with direct event execution or polling.

The sporadic server scheduling policy has been implemented as a commercial product in the run-time system of the Verdix Ada compiler. Many applications have also used a much less efficient application-level sporadic server. These real-time applications would benefit from a sporadic server scheduler implemented at the scheduler level.

B.13.3.3 Library-Level vs. Kernel-Level Implementation

The sporadic server interface described in this subclause requires the sporadic server policy to be implemented at the same level as the scheduler. In other words, the process sporadic server shall be implemented at the kernel level and the thread sporadic server policy shall be implemented at the same level as the thread scheduler, i.e., kernel or library level.

In an earlier interface for the sporadic server, this mechanism was implementable at a different level than the scheduler. This feature allowed the implementer to choose between an efficient scheduler-level implementation, or a simpler user or library-level implementation. However, the working group considered that this interface made the use of sporadic servers more complex and that library-level implementations would lack some of the important functionality of the sporadic server, namely the limitation of the actual execution time of aperiodic activities. The working group also felt that the interface described in this chapter does not preclude library-level implementations of threads intended to provide efficient low-overhead scheduling for threads that are not scheduled under the sporadic server policy.

B.13.3.4 Range of Scheduling Priorities

Each of the scheduling policies supported in POSIX.1b has an associated range of priorities. The priority ranges for each policy might or might not overlap with the priority ranges of other policies. For time-critical realtime applications it is usual for periodic and aperiodic activities to be scheduled together in the same processor. Periodic activities will usually be scheduled using the SCHED_FIFO scheduling policy, while aperiodic activities may be scheduled using SCHED_SPORADIC. Since the application developer will require complete control over the relative priorities of these activities in order to meet the application's timing requirements, it would be desirable for the priority ranges of SCHED_FIFO and SCHED_SPORADIC to overlap completely. Therefore, although the standard does not require any particular relationship between the different priority ranges, it is recommended that these two ranges should coincide.

B.13.3.5 Dynamically Setting the Sporadic Server Policy

Several members of the working group requested that implementations should not be required to support dynamically setting the sporadic server scheduling policy for a thread. The reason is that this policy may have a high overhead for library-level implementations of threads; and, if threads are allowed to dynamically set this policy, this overhead can be experienced even if the thread does not use that policy. By disallowing the dynamic setting of the sporadic server scheduling policy, these implementations can accomplish efficient scheduling for threads using other policies. If a strictly conforming application needs to use the sporadic server policy and is, therefore, willing to pay the overhead, it shall set this policy at the time of thread creation.

B.13.3.6 Limitation of the Number of Pending Replenishments

The number of simultaneously pending replenishment operations shall be limited for each sporadic server for two reasons: an unlimited number of replenishment operations would need an unlimited number of system resources to store all the pending replenishment operations; on the other hand, in some implementations each replenishment operation will represent a source of priority inversion (just for the duration of the replenishment operation) and thus, the maximum amount of replenishments shall be bounded to guarantee bounded response times. The way in which the number of replenishments is bounded is by lowering the priority of the sporadic server to *sched_ss_low_priority* when the number of pending replenishments has reached its limit. In this way, no new replenishments are scheduled until the number of pending replenishments decreases.

In the sporadic server scheduling policy defined in this standard, the application can specify the maximum number of pending replenishment operations for a single sporadic server, by setting the value of the *sched_ss_max_repl* scheduling parameter. This value shall be between one and {SS_REPL_MAX}, which is a maximum limit imposed by the implementation. The limit {SS_REPL_MAX} shall be greater than or equal to {_POSIX_SS_REPL_MAX}, which is defined to be four in this standard. The minimum limit of four was chosen so that an application can at least guarantee that four different aperiodic events can be processed during each interval of length equal to the replenishment period.

B.14 Clocks and Timers

⇒ **B.14 Clocks and Timers** *Add the following subclauses:*

B.14.3 Execution Time Monitoring

B.14.3.1 Introduction

The main goals of the execution time monitoring facilities defined in this chapter are to measure the execution time of processes and threads and to allow an application to establish CPU time limits for these entities. The analysis phase of time-critical realtime systems often relies on the measurement of execution times of individual threads or processes to determine whether the timing requirements will be met. Also, performance analysis techniques for soft deadline realtime systems rely heavily on the determination of these execution times. The execution time monitoring functions provide application developers with the ability to measure these execution times on line and open the possibility of dynamic execution-time analysis and system reconfiguration, if required. The second goal of allowing an application to establish execution time limits for individual processes or threads and detecting when they overrun allows program robustness to be increased by enabling on-line checking of the execution times. If errors are detected (possibly because of erroneous program constructs, the existence of errors in the analysis phase, or a burst of event arrivals) on-line detection and recovery are possible in a portable way. This feature can be extremely important for many time-critical applications. Other applications require trapping CPU-time errors as a normal way to exit an algorithm; for instance, some realtime artificial intelligence applications trigger a number of independent inference processes of varying accuracy and speed, limit how long they can run, and pick the best answer available when time runs out. In many periodic systems, overrun processes are simply restarted in the next resource period, after necessary end-of-period actions have been taken. This behavior allows algorithms that are inherently data-dependent to be made predictable.

The interface that appears in this chapter defines a new type of clock, the CPU-time clock, which measures execution time. Each process or thread can invoke the clock and timer functions defined in POSIX.1b to use them. Functions are also provided to access the CPU-time clock of other processes or threads to enable remote monitoring of these clocks. Monitoring of threads of other processes is not supported since these threads are not visible from outside of their own process with the interfaces defined in POSIX.1c.

B.14.3.2 Execution Time Monitoring Interface

The clock and timer interface defined in POSIX.1b (Section 14) only defines one clock, which measures wall-clock time. The requirements for measuring execution time of processes and threads, and setting limits to their execution time by detecting when they overrun, can be accomplished with that interface if a new kind of clock is defined. These new clocks measure execution time, and one is associated with each process and with each thread. The clock functions currently defined in

POSIX.1b can be used to read and set these CPU-time clocks, and timers can be created using these clocks as their timing base. These timers can then be used to send a signal when some specified execution time has been exceeded. The CPU-time clocks of each process or thread can be accessed by using the symbols CLOCK_PROCESS_CPUTIME_ID or CLOCK_THREAD_CPUTIME_ID.

The clock and timer interface defined in POSIX.1b and extended with the new kind of CPU-time clock would only allow processes or threads to access their own CPU-time clocks. However, many realtime systems require the possibility of monitoring the execution time of processes or threads from independent monitoring entities. In order to allow applications to construct independent monitoring entities that do not require cooperation from or modification of the monitored entities, two functions have been defined in this chapter: *clock_getcpuclockid*(), for accessing CPU-time clocks of other processes, and *pthread_getcpuclockid*(), for accessing CPU-time clocks of other threads. These functions return the clock identifier associated with the process or thread specified in the call. These clock IDs can then be used in the rest of the clock function calls.

The clocks accessed through these functions could also be used as a timing base for the creation of timers, thereby allowing independent monitoring entities to limit the CPU-time consumed by other entities. However, this possibility would imply additional complexity and overhead because of the need to maintain a timer queue for each process or thread to store the different expiration times associated with timers created by different processes or threads. The working group decided this additional overhead was not justified by application requirements. Therefore, creation of timers attached to the CPU-time clocks of other processes or threads has been specified as implementation defined.

B.14.3.3 Overhead Considerations

The measurement of execution time may introduce additional overhead in the thread scheduling, because of the need to keep track of the time consumed by each of these entities. In library-level implementations of threads, the efficiency of scheduling could be somehow compromised because of the need to make a kernel call, at each context switch, to read the process CPU-time clock. Consequently, a thread creation attribute called `cpu-clock-requirement` was defined to allow threads to disconnect their respective CPU-time clocks. However, the balloting group considered that this attribute itself introduced some overhead and that in current implementations it was not worth the effort. Therefore, the attribute was deleted, and thus thread CPU-time clocks are required for all threads if the Thread CPU-Time Clocks option is supported.

B.14.3.4 Accuracy of CPU-time Clocks

The mechanism used to measure the execution time of processes and threads is specified in this document as implementation defined. The reason for this requirement is that both the underlying hardware and the implementation architecture have a very strong influence on the accuracy achievable for measuring CPU-time. For some implementations, the specification of strict accuracy requirements would represent very large overheads or even the impossibility of being implemented.

Since the mechanism for measuring execution time is implementation defined, realtime applications will be able to take advantage of accurate implementations using a portable interface. Of course, strictly conforming applications cannot rely on any particular degree of accuracy, in the same way as they cannot rely on a very accurate measurement of wall clock time. There will always exist applications whose accuracy or efficiency requirements on the implementation are more rigid than the values defined in this or any other standard.

In any case, realtime applications would expect a minimum set of characteristics from most implementations. One such characteristic is that the sum of all the execution times of all the threads in a process equals the process execution time when no CPU-time clocks are disabled. This property may not always be true because implementations may differ in how they account for time during context switches. Another characteristic is that the sum of the execution times of all processes in a system equals the number of processors, multiplied by the elapsed time, assuming that no processor is idle during that elapsed time. However, in some systems it might not be possible to relate CPU-time to elapsed time. For example, in a heterogeneous multiprocessor system in which each processor runs at a different speed, an implementation may choose to define each "second" of CPU-time to be a certain number of "cycles" that a CPU has executed.

B.14.3.5 Existing Practice

Measuring and limiting the execution time of each concurrent activity are common features of most industrial implementations of realtime systems. Almost all critical realtime systems are currently built upon a cyclic executive. With this approach, a regular timer interrupt kicks off the next sequence of computations. It also checks that the current sequence has completed. If it has not, then some error recovery action can be undertaken (or at least an overrun is avoided). Current software engineering principles and the increasing complexity of software are driving application developers to implement these systems on multi-threaded or multi-process operating systems. Therefore, if a POSIX operating system is to be used for this type of application, then it must offer the same level of protection.

Execution time clocks are also common in most UNIX implementations, although these clocks usually have different requirements from those of realtime applications. The POSIX.1 *times*() function supports the measurement of the execution time of the calling process and its terminated child processes. This execution time is measured in clock ticks and is supplied as two different values with the user and system execution times, respectively. BSD {B60} supports the function *getrusage*(), which allows the calling process to get information about the resources used by itself and/or all of its terminated child processes. The resource usage includes user and system CPU time. Some UNIX systems have options to specify high resolution (up to one microsecond) CPU time clocks using the *times*() or the *getrusage*() functions.

The *times*() and *getrusage*() interfaces do not meet important realtime requirements such as the possibility of monitoring execution time from a different process or thread or the possibility of detecting an execution time overrun. The latter requirement is supported in some UNIX implementations that are able to send a signal when the execution time of a process has exceeded some specified value. For example, BSD defines the functions *getitimer*() and *setitimer*(), which can operate either on a realtime clock (wall clock) or on virtual-time or profile-time clocks, which measure CPU time in two different ways. These functions do not support access to the execution time of other processes. System V supports similar functions after release 4. Some emerging implementations of threads also support these functions.

IBM's MVS operating system supports per-process and per-thread execution time clocks. It also supports limiting the execution time of a given process.

Given all this existing practice, the working group considered that the POSIX.1b clocks and timers interface was appropriate to meet most of the requirements that realtime applications have for execution time clocks. Functions were added to get the CPU time clock IDs and to allow or disallow the thread CPU time clocks (in order to preserve the efficiency of some implementations of threads).

B.14.3.6 Clock Constants

The definition of the manifest constants CLOCK_PROCESS_CPUTIME_ID and CLOCK_THREAD_CPUTIME_ID allows processes or threads, respectively, to access their own execution-time clocks. However, given a process or thread, access to its own execution-time clock is also possible if the clock ID of this clock is obtained through a call to *clock_getcpuclockid*() or *pthread_getcpuclockid*(). Therefore, these constants are not necessary and could be deleted to make the interface simpler. Their existence saves one system call in the first access to the CPU-time clock of each process or thread. The working group considered this issue and decided to leave the constants in the standard because they are closer to the POSIX.1b use of clock identifiers.

B.14.3.7 Library Implementations of Threads

In library implementations of threads, kernel entities and library threads can coexist. In this case, if the CPU-time clocks are supported, most of the clock and timer functions will need to have two implementations: one in the thread library and one in the system calls library. The main difference between these two implementations is that the thread library implementation will have to deal with clocks and timers that reside in the thread space, while the kernel implementation will operate on timers and clocks that reside in kernel space. In the library implementation, if the clock ID refers to a clock that resides in the kernel, a kernel call will have to be made. The correct version of the function can be chosen by specifying the appropriate order for the libraries during the link process.

B.14.3.8 History of Resolution Issues: Deletion of the `enable` attribute

In the draft corresponding to the first balloting round, CPU-time clocks had an attribute called `enable`. This attribute was introduced by the working group to allow implementations to avoid the overhead of measuring execution time for processes or threads for which this measurement was not required. However, the `enable` attribute received several ballot objections. The main objection was that processes are already required to measure execution time by the POSIX.1 *times*() function. Consequently, the `enable` attribute was considered unnecessary and was deleted from this standard.

B.14.4 Rationale Relating to Timeouts

B.14.4.1 Requirements for Timeouts

Realtime systems that have to operate reliably over extended periods without human intervention are characteristic in embedded applications such as avionics, machine control, and space exploration, as well as more mundane applications such as cable TV, security systems, and plant automation. A multi-tasking paradigm, in which many independent and/or cooperating software functions relinquish the processor(s) while waiting for a specific stimulus, resource, condition, or operation completion, is very useful in producing well-engineered programs for such systems. For such systems to be robust and fault tolerant, expected occurrences that are unduly delayed or that never occur must be detected so that appropriate recovery actions may be taken. This requirement is difficult to achieve if there is no way for a task to regain control of a processor once it has relinquished control (blocked) awaiting an occurrence which, perhaps because of corrupted code, hardware malfunction, or latent software bugs, will not happen when expected. Therefore, the common practice in realtime operating systems is to provide a capability to time out such blocking services. Although there are several methods already defined by POSIX to achieve this timeout capability, none is as reliable or efficient as initiating a timeout simultaneously with initiating a blocking service. Timeouts are especially critical in hard-realtime embedded systems because the processors typically have little time reserve, and allowed fault recovery times are measured in milliseconds rather than seconds.

The working group largely agreed that such timeouts were necessary and ought to become part of the standard, particularly vendors of realtime operating systems whose customers had already expressed a strong need for timeouts. There was some resistance to inclusion of timeouts in the standard because the desired effect, fault tolerance, could, in theory, be achieved using existing facilities and alternative software designs, but there was no compelling evidence that realtime system designers would embrace such designs at the sacrifice of performance and/or simplicity.

B.14.4.2 Which Services Should Be Timed Out?

Originally, the working group considered the prospect of providing timeouts on *all* blocking services, including those currently existing in POSIX.1, POSIX.1b, and POSIX.1c, and future interfaces to be defined by other working groups, as a general policy. This proposal was rather quickly rejected because of the scope of such a change, and the fact that many of those services would not normally be used in a realtime context. More traditional time-sharing solutions to time out would suffice for most of the POSIX.1 interfaces, while others had asynchronous alternatives that, while more complex to utilize, would be adequate for some realtime and all nonrealtime applications.

The list of potential candidates for timeouts was narrowed to the following for further consideration:

POSIX.1b

- *sem_wait*()
- *mq_receive*()
- *mq_send*()
- *lio_listio*()
- *aio_suspend*()
- *sigwait*()
  timeout already implemented by *sigtimedwait*()

POSIX.1c

- *pthread_mutex_lock*()
- *pthread_join*()
- *pthread_cond_wait*()
  timeout already implemented by *pthread_cond_timedwait*()

POSIX.1

- *read*()
- *write*()

After further review by the working group, the *read*(), *write*(), and *lio_listio*() functions (all forms of blocking synchronous I/O) were eliminated from the list because

(1) Asynchronous alternatives exist,

(2) Timeouts can be implemented, albeit nonportably, in device drivers, and

(3) A strong desire existed not to introduce modifications to POSIX.1 interfaces.

The working group ultimately rejected *pthread_join*() since both that interface and a timed variant of that interface are nonminimal and may be implemented as a library function. See B.14.4.3 for a library implementation of *pthread_join*().

Thus there was a consensus among the working group members to add timeouts to 4 of the remaining 5 functions (the timeout for *aio_suspend*() was ultimately added directly to POSIX.1b, while the others are added here in POSIX.1d). However, *pthread_mutex_lock*() remained contentious.

Many balloting group members feel that *pthread_mutex_lock*() falls into the same class as the other functions; that is, it is desirable to time out a mutex lock because a mutex may fail to be unlocked due to errant or corrupted code in a critical section (looping or branching outside of the unlock code) and, therefore, is equally in need of a reliable, simple, and efficient timeout. In fact, since mutexes are intended to guard small critical sections, most *pthread_mutex_lock*() calls would be expected to obtain the lock without blocking nor utilizing any kernel service, even in implementations of threads with global contention scope; the timeout alternative need only be considered after it is determined that the thread shall block.

Those opposed to timing out mutexes feel that the very simplicity of the mutex is compromised by adding a timeout semantic and that to do so is senseless. They claim that if a timed mutex is really deemed useful by a particular application, then it can be constructed from the facilities already in POSIX.1b and POSIX.1c. The two C language library implementations of mutex locking with timeout in Figure B-4 and Figure B-5 represent the solutions offered (in both implementations, the timeout parameter is specified as absolute time, not relative time as in the proposed POSIX.1c interfaces):

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

int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *timeout)
        {
        struct timespec timenow;

        while (pthread_mutex_trylock(mutex) == EBUSY)
                {
                clock_gettime(CLOCK_REALTIME, &timenow);
                if (timespec_cmp(&timenow,timeout) >= 0)
                        {
                        return ETIMEDOUT;
                        }
                pthread_yield();
                }
        return 0;
        }
```

Figure B-4 – Spinlock Implementation

The spinlock implementation is generally unsuitable for any application using priority based thread scheduling policies such as {SCHED_FIFO} or {SCHED_RR}. The reason is that the mutex could currently be held by a thread of lower priority within the same allocation domain; but, since the waiting thread never blocks, only threads of equal or higher priority will ever run. Therefore, the mutex cannot be unlocked. Setting priority inheritance or priority ceiling protocol on the mutex does not solve this problem, since the priority of a mutex-owning thread is only boosted if higher priority threads are blocked waiting for the mutex, clearly not the case for this spinlock.

The condition wait implementation effectively substitutes the *pthread_cond_timedwait*() function (which is currently timed out) for the desired *pthread_mutex_timedlock*(). Since waits on condition variables currently do not include protocols that avoid priority inversion, this method is generally unsuitable for realtime applications because it does not provide the same priority inversion protection as the untimed *pthread_mutex_lock*(). Also, for any given implementations of the current mutex and condition variable primitives, this library implementation has a performance cost at least 2.5 times that of the untimed *pthread_mutex_lock*() even in the case where the timed mutex is readily locked without blocking. Even in uniprocessors or where assignment is atomic, at least an additional *pthread_cond_signal*() is required. In this case,

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

struct timed_mutex
        {
        int locked;
        pthread_mutex_t mutex;
        pthread_cond_t cond;
        };
typedef struct timed_mutex timed_mutex_t;

int timed_mutex_lock(timed_mutex_t *tm,
                    const struct timespec *timeout)
        {
        int timedout=FALSE;
        int error_status;

        pthread_mutex_lock(&tm->mutex);

        while (tm->locked && !timedout)
                {
                if ((error_status=pthread_cond_timedwait(&tm->cond,
                                                    &tm->mutex,
                                                    timeout))!=0)
                        {
                        if (error_status==ETIMEDOUT) timedout = TRUE;
                        }
                }

        if(timedout)
                {
                pthread_mutex_unlock(&tm->mutex);
                return ETIMEDOUT;
                }
        else
                {
                tm->locked = TRUE;
                pthread_mutex_unlock(&tm->mutex);
                return 0;
                }
        }

void timed_mutex_unlock(timed_mutex_t *tm)
        {
        pthread_mutex_lock(&tm->mutex); /*for case assignment not atomic*/
        tm->locked = FALSE;
        pthread_mutex_unlock(&tm->mutex);
        pthread_cond_signal(&tm->cond);
        }
```

Figure B-5 – Condition Wait Implementation

pthread_mutex_timedlock() could be implemented at effectively no performance penalty because the timeout parameters need only be considered after it is determined that the mutex cannot be locked immediately.

Thus it has not yet been shown that the full semantics of mutex locking with timeout can be efficiently and reliably achieved using existing interfaces. Even if the existence of an acceptable library implementation were proven, it is difficult to justify why the *interface* itself should not be made portable, especially considering approval for the other four timeouts.

B.14.4.3 Rationale for Library Implementation of *pthread_timedjoin*()

The *pthread_join*() C Language example shown in Figure B-6 demonstrates that it is possible, using existing *pthread* facilities, to construct a variety of thread that allows for joining such a thread, but allows the *join* operation to time out. This behavior is achieved by using a *pthread_cond_timedwait*() to wait for the thread to exit. A small *timed_thread* descriptor structure is used to pass parameters from the creating thread to the created thread and from the exiting thread to the joining thread. This implementation is roughly equivalent to what a normal *pthread_join*() implementation would do, with the single change being that *pthread_cond_timedwait*() is used in place of a simple *pthread_cond_wait*().

Since it is possible to implement such a facility entirely from existing *pthread* interfaces and with roughly equal efficiency and complexity to an implementation that would be provided directly by a *pthreads* implementation, it was the consensus of the working group members that any *pthread_timedjoin*() facility would be unnecessary and should not be provided.

```
/*
 * Construct a thread variety entirely from existing functions
 * with which a join can be done, allowing the join to time out.
 */

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

struct timed_thread {
    pthread_t t;
    pthread_mutex_t m;
    int exiting;
    pthread_cond_t exit_c;
    void *(*start_routine)(void *arg);
    void *arg;
    void *status;
};

typedef struct timed_thread *timed_thread_t;
static pthread_key_t timed_thread_key;
static pthread_once_t timed_thread_once = PTHREAD_ONCE_INIT;

static void timed_thread_init()
{
    pthread_key_create(&timed_thread_key, NULL);
}

static void *timed_thread_start_routine(void *args)

/*
 * Routine to establish thread specific data value and run the actual
 * thread start routine which was supplied to timed_thread_create().
 */
```

```
{
    timed_thread_t tt = (timed_thread_t) args;

    pthread_once(&timed_thread_once, timed_thread_init);
    pthread_setspecific(timed_thread_key, (void *)tt);
    timed_thread_exit((tt->start_routine)(tt->arg));
}

int timed_thread_create(timed_thread_t ttp, const pthread_attr_t *attr,
                void *(*start_routine)(void *), void *arg)

/*
 * Allocate a thread which can be used with timed_thread_join().
 */

{
    timed_thread_t tt;
    int result;

    tt = (timed_thread_t) malloc(sizeof(struct timed_thread));
    pthread_mutex_init(&tt->m,NULL);
    tt->exiting = FALSE;
    pthread_cond_init(&tt->exit_c,NULL);
    tt->start_routine = start_routine;
    tt->arg = arg;
    tt->status = NULL;

    if ((result = pthread_create(&tt->t, attr,
                    timed_thread_start_routine, (void *)tt)) != 0) {
        free(tt);
        return result;
    }

    pthread_detach(tt->t);
    ttp = tt;
    return 0;
}

timed_thread_join(timed_thread_t tt,
                  struct timespec *timeout,
                  void **status)
{
    int result;

    pthread_mutex_lock(&tt->m);
    result = 0;
    /*
     * Wait until the thread announces that it's exiting, or until timeout.
     */
    while (result == 0 && ! tt->exiting) {
        result = pthread_cond_timedwait(&tt->exit_c, &tt->m, timeout);
    }
    pthread_mutex_unlock(&tt->m);
    if (result == 0 && tt->exiting) {
        *status = tt->status;
        free((void *)tt);
        return result;
    }
    return result;
}
```

```
timed_thread_exit(void *status)
{
    timed_thread_t tt;
    void *specific;

    if ((specific=pthread_getspecific(timed_thread_key)) == NULL){
        /*
         * Handle cases which won't happen with correct usage.
         */
        pthread_exit(NULL);
    }
    tt = (timed_thread_t) specific;
    pthread_mutex_lock(&tt->m);
    /*
     * Tell a joiner that we're exiting.
     */
    tt->status = status;
    tt->exiting = TRUE;
    pthread_cond_signal(&tt->exit_c);
    pthread_mutex_unlock(&tt->m);
    /*
     * Call pthread exit() to call destructors and really exit the thread.
     */
    pthread_exit(NULL);
}
```

Figure B-6 – ***pthread_join*****() with timeout**

B.14.4.4 Form of the Timeout Interfaces

The working group considered a number of alternative ways to add timeouts to blocking services. At first, a system interface that would specify a one-shot or persistent timeout to be applied to subsequent blocking services invoked by the calling process or thread was considered because it allowed all blocking services to be timed out in a uniform manner with a single additional interface; this interface was rather quickly rejected because it could easily result in the wrong services being timed out.

It was suggested that a timeout value might be specified as an attribute of the object (e.g., semaphore, mutex, message queue), but there was no consensus on this suggestion, either on a case-by-case basis or for all timeouts.

Looking at the two existing timeouts for blocking services indicates that the working group members favor a separate interface for the timed version of a function. However, *pthread_cond_timedwait*() utilizes an absolute timeout value while *sigtimedwait*() uses a relative timeout value. The working group members agreed that relative timeout values are appropriate where the timeout mechanism's primary use was to deal with an unexpected or error situation, but they are inappropriate when the timeout has to expire at a particular time or before a specific deadline. For the timeouts being introduced in this document, the working group considered allowing both relative and absolute timeouts as is done with POSIX.1b timers, but ultimately favored the simpler absolute timeout form.

An absolute time measure can be easily implemented on top of an interface that specifies relative time by reading the clock, calculating the difference between the current time and the desired wake up time, and issuing a relative timeout call. But there is a race condition with this approach because the thread could be preempted after reading the clock, but before making the timed out call; in this case, the thread would be awakened later than it should and, thus, if the wake up time represented a deadline, the thread would miss it.

There is also a race condition when trying to build a relative timeout on top of an interface that specifies absolute timeouts. In this case, the clock would have to be read to calculate the absolute wake up time as the sum of the current time plus the relative timeout interval. In this case, if the thread is preempted after reading the clock, but before making the timed out call, the thread would be awakened earlier than desired.

But the race condition with the absolute timeouts interface is not as bad as the one that happens with the relative timeout interface because there are simple workarounds. For the absolute timeouts interface, if the timing requirement is a deadline, it can still be met because the thread woke up earlier than the deadline. If the timeout is just used as an error recovery mechanism, the precision of timing is not really important. If the timing requirement is that between actions A and B a minimum interval of time must elapse, the absolute timeout interface can be safely used by reading the clock after action A has been started. It could be argued that, since the call with the absolute timeout is atomic from the application point of view, it is not possible to read the clock after action A if this action is part of the timed out call. But for the calls for which timeouts are specified (e.g., locking a mutex, waiting for a semaphore, waiting for a message, waiting until there is space in a message queue), the timeouts that an application would build on these actions would not be triggered by these actions themselves, but by some other external action. For example, to wait for at least 20 milliseconds for a message to arrive to a message queue, this time interval would be started by some event that would trigger both the action that produces the message and the action that waits for the message to arrive, and not by the wait-for-message operation itself. In this case, the workaround proposed above could be used.

For these reasons, the absolute timeout is preferred over the relative timeout interface.

⇒ **Annex B Rationale and Notes** *Add the following subclause.*

B.19 Advisory Information

The POSIX.1b standard contains an informative annex with proposed interfaces for “realtime files.” These interfaces could determine groups of the exact parameters required to do “direct I/O” or “extents.” These interfaces were objected to by a a significant portion of the balloting group as too complex. A portable application had little chance of correctly navigating the large parameter space to match its desires to the system. In addition, they only applied to a new type of file (realtime files) and they told the implementation exactly what to do as opposed to advising the implementation on application behavior and letting it optimize for the system on which the (portable) application was running. For example, it was not clear how a system that had a disk array should set its parameters.

There seemed to be several overall goals:

— Optimizing Sequential Access

— Optimizing Caching Behavior

— Optimizing I/O data transfer

— Preallocation

The advisory interfaces, *posix_fadvise*() and *posix_madvise*(), satisfy the first two goals. The POSIX_FADV_SEQUENTIAL and POSIX_MADV_SEQUENTIAL *advice* tells the implementation to expect serial access. Typically the system will prefetch the next several serial accesses in order to overlap I/O. It may also free previously accessed serial data if memory is tight. If the application is not doing serial access, it can use POSIX_FADV_WILLNEED and POSIX_MADV_WILLNEED to accomplish I/O overlap, as required. When the application advises POSIX_FADV_RANDOM or POSIX_MADV_RANDOM behavior, the implementation usually tries to fetch a minimum amount of data with each request; and it does not expect much locality. POSIX_FADV_DONTNEED and POSIX_MADV_DONTNEED allow the system to free up caching resources as the data will not be required in the near future.

POSIX_FADV_NOREUSE tells the system that caching the specified data is not optimal. For file I/O, the transfer should go directly to the user buffer instead of being cached internally by the implementation. To portably perform direct disk I/O on all systems, the application shall perform its I/O transfers according to the following rules:

(1) The user buffer should be aligned according to the {POSIX_REC_XFER_-ALIGN} *pathconf*() variable.

(2) The number of bytes transferred in an I/O operation should be a multiple of the {POSIX_ALLOC_SIZE_MIN} *pathconf*() variable.

(3) The offset into the file at the start of an I/O operation should be a multiple of the {POSIX_ALLOC_SIZE_MIN} *pathconf*() variable.

(4) The application should ensure that all threads that open a given file specify POSIX_FADV_NOREUSE to be sure that there is no unexpected interaction between threads using buffered I/O and threads using direct I/O to the same file.

In some cases, a user buffer should be properly aligned in order to be transferred directly to/from the device. The {POSIX_REC_XFER_ALIGN} *pathconf*() variable tells the application the proper alignment.

The preallocation goal is met by the space control function, *posix_fallocate*(). The application can use *posix_fallocate*() to guarantee no [ENOSPC] errors and to improve performance by prepaying any overhead required for block allocation.

Implementations may use information conveyed by a previous *posix_fadvise*() call to influence the manner in which allocation is performed. For example, assume an application does the following calls:

```
fd = open("file")
posix_fadvise(fd, offset, len, POSIX_FADV_SEQUENTIAL)
posix_fallocate(fd, len, size)
```

As a result, an implementation might allocate the file contiguously on disk.

Finally, the *pathconf*() variables {POSIX_REC_MIN_XFER_SIZE}, {POSIX_REC_MAX_XFER_SIZE} and {POSIX_REC_INCR_XFER_SIZE} tell the application a range of transfer sizes that are recommended for best I/O performance.

Where bounded response time is required, the vendor can supply the appropriate settings of the advisories to achieve a guaranteed performance level.

The interfaces meet the goals while allowing applications using regular files to take advantage of performance optimizations. The interfaces tell the implementation expected application behavior that the implementation can use to optimize performance on a particular system with a particular dynamic load.

The *posix_memalign*() function was added to allow for the allocation of specifically aligned buffers, e.g. for {POSIX_REC_XFER_ALIGN}.

The working group also considered the alternative of adding a function that would return an aligned pointer to memory within a user supplied buffer. This method was not considered to be best because it potentially wastes large amounts of memory when buffers need to be aligned on large alignment boundaries.

Identifier Index

Alphabetic Topical Index

A

B

C

D

E

F

G

H

I

J

K

L

R

S

T

U

V

W