

STANDARDS PROJECT
Information Technology—Portable
Operating System Interface (POSIX®)—
Part 1:
System Application Program Interface
(API)—Amendment x:
Additional Realtime Extensions [C Language]

Sponsor
Portable Application Standards Committee
of the
IEEE Computer Society

Work Item Number: JTC1 22.40

Abstract: (IEEE Std P1003.1d-199x) 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.

POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

P1003.1d / D14
July 1999

Copyright © 1999 by the Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017, USA
All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change. Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of IEEE standardization activities. Permission is also granted for member bodies and technical committees of ISO and IEC to reproduce this document for purposes of developing a national position.

Other entities seeking permission to reproduce this document for standardization or other activities, or to reproduce portions of this document for these or other uses, must contact the IEEE Standards Department for the appropriate license. Use of information contained in this unapproved draft is at your own risk.

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (732) 562-3800
+1 (732) 562-1571 [FAX]
July 1999

SH XXXXX

Editor's Notes

In addition to your paper ballot, you are also asked to e-mail any balloting comments: please read the balloting instructions and the cover letter for the ballot that accompanied this draft.

This section will not appear in the final document. It is used for introductory editorial comments concerning this draft. Additional Editor's Notes are scattered throughout the document.

This draft uses small numbers or letters in the right margin in lieu of change bars. "E" denotes changes from Draft 13 to Draft 14. Trivial informative (i.e., non-normative) changes and purely editorial changes such as grammar, spelling, or cross references are not diff-marked. Changes of function names are not diff-marked either. Since this is a recirculation draft, only normative text marked with "E" is open for comments in this ballot. Revision indicators prior to "c" have been removed from this draft.

Since 1998 there is a new backwards compatibility requirement on the amendments to the base POSIX.1 standard, which states that the base standard will not be changed in such a way as to cause implementations or strictly conforming applications to no longer conform. The implications of this requirement are that no new interface specifications can be included that are not under an option; and that names for new interfaces must begin or end with one of the reserved prefixes or suffixes, including those defined in POSIX.1a. This standard incorporates the required changes since Draft 11.

Until draft 12, the rationale text for most of the sections had been temporarily moved from Annex B and interspersed with the appropriate sections. This collocation of rationale with its accompanying text was done to encourage the Technical Reviewers to maintain the rationale text, as well as provide explanations to the reviewers and balloters. However, for the last recirculations, since draft 13 all rationale subclauses have been moved back to Annex B.

Please report typographical errors to:

Michael González Harbour
Dpto. de Electrónica y Computadores
Universidad de Cantabria
Avenida de los Castros s/n
39005 - Santander
SPAIN
TEL: +34 942 201483
FAX: +34 942 201402
Email: mgh@ctr.unican.es

(Electronic mail is preferred.)

The copying and distribution of IEEE balloting drafts is accomplished by the Standards Office. To report problems with reproduction of your copy, or to request additional copies of this draft, contact:

Tracy Woods
IEEE Computer Society,
1730 Massachusetts Avenue, NW,
Washington DC 20036-1992.
Phone: +1-202-371-1013
Fax: +1-202-728-0884
E-mail: twoods@computer.org
Web page: <http://www.computer.org/standard/draftstd.htm>

POSIX.1d Change History

This section is provided to track major changes between drafts.

Draft 14 [July 1999] Third recirculation of new ballot.

- *Only editorial changes have been introduced. The main change was to align the text used for optional features with the text used in POSIX.1b and POSIX.1c.*

Draft 13 [April 1999] Second recirculation of new ballot.

- *Annex I (Device Control) and Annex K (Balloting Instructions) were removed from the draft.*
- *Moved the “Conventions” and “Normative References” subclauses into these editor’s notes, because no amendment to the equivalent subclauses in POSIX.1 was intended.*
- *The posix_spawn() functions now use a special value as the exit code to indicate that a child process had been created, but the spawn operation was unsuccessful.*
- *The spawn attributes datatype was changed from a structure to an opaque type with associated functions to initialize and destroy the object, and to set and get each individual attribute.*

Draft 12 [January 1999] First recirculation of new ballot.

- *Annex J, Interrupt control was removed from the draft.*
- *Removed the CPU-clock-requirement thread creation attribute.*
- *Removed the timeout-allowed mutex creation attribute.*
- *Moved Section 20 into Section 14, Clocks and Timers, as a new subsection.*
- *Converted timeouts to absolute values, instead of relative intervals.*
- *Better description of optional features to follow the new backwards compatibility requirement.*
- *Changed the posix_spawn() interface to better match the POSIX.5 POSIX_Process_Primitives.Start_Process*
- *Made posix_fallocate() be able to change the size of the file.*

Draft 11 [May 1998] New Ballot

- *A new ballot group was formed and a new ballot process was started. This implied removing annexes G and H, which were associated with the previous ballot process.*
- *Many function and structure names were changed according to the new backwards compatibility requirement.*

- *All the text related to interrupt control and device control was moved to the appropriate annexes.*
 - *The annex on performance metrics was removed, because it was outdated.*
 - *A new mutex creation attribute was added to enable or disable the use of the timed wait operation on individual mutexes.*
 - *The functions to spawn a process were aligned to the similar procedure for starting a process in IEEE 1003.5:1992.*
- Draft 10.0 [January 1997] Final re-circulation.*
- *Minor clean-up.*
 - *Former Section 23 renumbered to Section 21.*
 - *Added Annex G (Unresolved Objections).*
 - *Added Annex H (Objection Status).*
- Draft 9.0 [September 1996] Internal re-circulation.*
- *Move Section 21 (Device Control) to Annex C.*
 - *Move Section 22 (Interrupt Control) to Annex D.*
 - *Addition of stubs.*
- Draft 8.06 [July 1995] Internal re-circulation.*
- *Changes resulting from ballot resolution.*
 - *Dropping of Section 22.*
- Draft 8 [September 1993] First Ballot*
- *Editorial changes from review of Draft 7.5.*
- Draft 7.5 [August 1993] Second Mock Ballot*
- *Document converted from LIS & C Binding to 'Thick' C as a result of working group decision to drop LIS work.*
 - *Document put into amendment form for merged 9945-1, POSIX.1b & POSIX.1c.*
- Draft 7 [June 1993]*
- *Removal of both LIS and C binding Test Assertions from the document. These sections have been archived for future use.*
 - *This draft has been reorganized into functional groupings, following the reorganization of P1003.1(LIS).*
 - *Performance metrics were moved to Annex G.*
 - *Changes as a result of mock ballot of draft 6 were incorporated and reviewed by the working group.*

Draft 6 [February 1993] Mock Ballot

- *The Process Primitives (C) & (LIS) sections were updated. Rationale from previous 'thick' C section was inserted.*
- *Additional Test Assertion text was added; however there is more work to be done.*

Draft 5 [December 1992]

- *Interrupt Control (LIS) and (C Binding) sections were added.*
- *Test Assertion text was added to sections 5, 6, 9, and 17.*
- *The document was restructured to provide sections for LIS, C Bindings, Test Assertions (LIS) and Test Assertions (C Bindings).*

Draft 5	Draft 4	Section Name
1	1	General
2	2	Terminology and General Requirements
3	3	Process Primitives(LIS)
4	—	Process Primitives(C)
5	—	Process Primitives Test Assertions(LIS)
6	—	Process Primitives Test Assertions(C)
7	5	Timeouts for Blocking Services(LIS)
8	6	Timeouts for Blocking Services(C)
9	—	Timeouts for Blocking Services Test Assertions(LIS)
10	—	Timeouts for Blocking Services Test Assertions(C)
11	7	Execution Time Monitoring(LIS)
12	8	Execution Time Monitoring(C)
13	—	Execution Time Monitoring Test Assertions(LIS)
14	—	Execution Time Monitoring Test Assertions(C)
15	9	Sporadic Server(LIS)
16	10	Sporadic Server(C)
17	—	Sporadic Server Test Assertions(LIS)
18	—	Sporadic Server Test Assertions(C)
19	11	Device Control(LIS)
20	12	Device Control(C)
21	—	Device Control Test Assertions(LIS)
22	—	Device Control Test Assertions(C)
23	—	Interrupt Control(LIS)
24	—	Interrupt Control(C)
25	—	Interrupt Control Test Assertions(LIS)
26	—	Interrupt Control Test Assertions(C)

Draft 4 [September 1992]

- *Signal disposition parameters were added to the "Process Primitives" section (4).*
- *The "Timeouts On Blocking Services" section was replaced with a Language-independent section (5) and a C Binding section (6).*

- The "Execution Time Monitoring" section was replaced with a Language-independent section (7) and a C Binding section (8).
- The "Sporadic Server" section was replaced with a Language-independent section (9) and a C Binding section (10).
- A new Language-independent version of "Device Control" was added as section (11).
- A new C Binding version of "Device Control" was added as section (12).
- Test Assertions are still to be added.

Draft 3 [May 1992]

- Corrections and editorial changes were made to the Process Primitives section (3).
- The CPU Time Clock section (5) was added.
- The Sporadic Server section (6) was added.
- Due to a system crash, some of the updates to sections for this draft may have been lost. If discrepancies are noted please contact the editor.

Draft 2 [February 1992]

- The Process Primitives section was moved from Annex A to Section 3.
- The Timeout Facilities section was moved from Annex B to Section 4.
- Section 4 was cleaned up.

Draft 1 [November 1991]

- The first draft of the `posix_spawn()` function was added to the draft as Annex A.
- The first draft of the `timeout facilities` was added to the draft as Annex B.

Normative References

NOTE: This standard does not amend subclause 1.2, Normative References, of ISO/IEC 9945-1:1996. However, the Normative References of ISO/IEC 9945-1:1996 are repeated here for information. In addition, since IEEE P1003.1d modifies ISO/IEC 9945-1:1996, we have included the latter among this informal list of references.

The following standards contain provisions which, through references in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed

below. Members of IEC and ISO maintain registers of currently valid International Standards.

- {1} ISO/IEC 9899: 1995¹⁾, *Information processing systems—Programming languages—C*.
- {2} 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]*.
- {3} IEEE Std 610-1990, *IEEE Standard Computer Dictionary — A Compilation of IEEE Standard Computer Glossaries*

Conventions

NOTE: This standard does not amend subclause 2.1, Conventions, of ISO/IEC 9945-1:1996. However, we repeat this subclause here for information.

This document uses the following typographic conventions:

- (1) The *italic* font is used for:
 - Cross references to defined terms within 2.2.1 and 2.2.2; symbolic parameters that are generally substituted with real values by the application
 - C language data types and function names (except in function Synopsis subclauses)
 - Global external variable names
 - Function families; references to groups of closely related functions (such as *directory()*, *exec()*, etc.)
- (2) The **bold** font is used with a word in all capital letters, such as
PATH
to represent an environment variable. It is also used for the term “**NULL pointer.**”
- (3) The constant-width (Courier) font is used:
 - For C language data types and function names within function Synopsis subclauses
 - To illustrate examples of system input or output where exact usage is depicted
 - For references to utility names and C language headers

1) ISO/IEC documents can be obtained from the ISO office, 1, rue de Varembe, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse. ISO publications are also available in the United States from the Sales Department, American National Standards Institute, 11 West 42nd Street, 13th Floor, New York, NY 10036, USA.

— For names of attributes in attributes objects

- (4) Symbolic constants returned by many functions as error numbers are represented as:

[ERRNO]

See 2.4.

- (5) Symbolic constants or limits defined in certain headers are represented as

{OPEN_MAX}

See 2.8 and 2.9.

In some cases tabular information is presented “inline”; in others it is presented in a separately labeled table. This arrangement was employed purely for ease of typesetting and there is no normative difference between these two cases.

The conventions listed previously are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

NOTEs provided as parts of labeled tables and figures are integral parts of this standard (normative). Footnotes and notes within the body of the text are for information only (informative).

Numerical quantities are presented in international style: comma is used as a decimal sign and units are from the International System (SI).

POSIX.1d Ballot Coordinator

The ballot coordinator for POSIX.1d is Jim Oblinger. During balloting he is the person who coordinates the resolution process and resolves procedural issues.

POSIX.1d Technical Reviewers

The individuals denoted in Table i are the Technical Reviewers for this draft. During balloting they are the subject matter experts who coordinate the resolution process for specific sections, as shown.

Table i — POSIX.1j Technical Reviewers

Section	Description	Reviewer
	<i>Ballot Coordinator</i>	Jim Oblinger
3	<i>Spawn a Process</i>	Frank Prindle
6,11,15	<i>Timeouts</i>	Michael González
13	<i>Sporadic Server Scheduling</i>	Michael González
14,16,20	<i>Execution Time Monitoring</i>	Michael González
21	<i>Advisory Information</i>	Karen Gordon

Contents

	PAGE
Introduction	v
Section 1: General	1
1.1 Scope	1
1.3 Conformance	3
Section 2: Terminology and General Requirements	5
2.2 Definitions	5
2.3 General Concepts	6
2.7 C Language Definitions	7
2.8 Numerical Limits	8
2.9 Symbolic Constants	10
Section 3: Process Management	13
3.1 Process Creation and Execution	13
3.1.1 Process Creation	13
3.1.2 Execute a File	13
3.1.4 Spawn File Actions	14
3.1.5 Spawn Attributes	16
3.1.6 Spawn a Process	21
3.2 Process Termination	26
3.2.1 Wait for Process Termination	26
Section 4: Process Environment	27
4.8 Configurable System Variables	27
4.8.1 Get Configurable System Variables	27
Section 5: Files and Directories	29
5.7 Configurable Pathname Variables	29
5.7.1 Get Configurable Pathname Variables	29
Section 6: Input and Output Primitives	31
6.7 Asynchronous Input and Output	31
6.7.1 Data Definitions for Asynchronous Input and Output	31
Section 11: Synchronization	33
11.2 Semaphore Functions	33
11.2.6 Lock a Semaphore	33
11.2.7 Unlock a Semaphore	35
11.3 Mutexes	35
11.3.3 Locking and Unlocking a Mutex	35

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 13: Execution Scheduling	39
13.1 Scheduling Parameters	39
13.2 Scheduling Policies	39
13.2.3 SCHED_OTHER	40
13.2.4 SCHED_SPORADIC	40
13.3 Process Scheduling Functions	42
13.3.1 Set Scheduling Parameters	42
13.3.3 Set Scheduling Policy and Scheduling Parameters	43
13.4 Thread Scheduling	43
13.4.1 Thread Scheduling Attributes	43
13.4.3 Scheduling Allocation Domain	44
13.4.4 Scheduling Documentation	44
13.5 Thread Scheduling Functions	44
13.5.1 Thread Creation Scheduling Attributes	44
13.5.2 Dynamic Thread Scheduling Parameters Access	45
 Section 14: Clocks and Timers	 47
14.2 Clock and Timer Functions	47
14.2.1 Clocks	47
14.2.2 Create a Per-Process Timer	48
14.3 Execution Time Monitoring	48
14.3.1 CPU-time Clock Characteristics	49
14.3.2 Accessing a Process CPU-time Clock	49
14.3.3 Accessing a Thread CPU-time Clock	50
 Section 15: Message Passing	 53
15.2 Message Passing Functions	53
15.2.4 Send a Message to a Message Queue	53
15.2.5 Receive a Message from a Message Queue	55
 Section 16: Thread Management	 59
16.1 Threads	59
16.2.2 Thread Creation	59
 Section 18: Thread Cancellation	 61
18.1 Thread Cancellation Overview	61
18.1.2 Cancellation Points	61
 Section 20: Advisory Information	 63
20.1 I/O Advisory Information and Space Control	63
20.1.1 File Advisory Information	63
20.1.2 File Space Control	64
20.2 Memory Advisory Information and Alignment Control	66
20.2.1 Memory Advisory Information	66
20.2.2 Aligned Memory Allocation	68
 Annex A (informative) Bibliography	 71
A.2 Other Standards	71
A.3 Historical Documentation and Introductory Texts	71

Annex B (informative) Rationale and Notes	73
B.2 Definitions and General Requirements	73
B.3 Process Primitives	73
B.13 Execution Scheduling	89
B.14 Clocks and Timers	93
B.20 Advisory Information	106

Identifier Index	109
----------------------------	-----

Alphabetic Topical Index	111
------------------------------------	-----

FIGURES

Figure B-1 – <i>posix_spawn()</i> Equivalent	89
Figure B-2 – I/O Redirection with <i>posix_spawn()</i>	89
Figure B-3 – Spawning a new Userid Process	89
Figure B-4 – Spinlock Implementation	100
Figure B-5 – Condition Wait Implementation	101
Figure B-6 – <i>pthread_join()</i> with timeout	105

TABLES

Table 2-4 – Optional Minimum Values	8
Table 2-6 – Optional Run-Time Invariant Values (Possibly Indeterm.)	9
Table 2-7 – Optional Pathname Variable Values	10
Table 2-11 – Versioned Compile-Time Symbolic Constants	11
Table 4-3 – Optional Configurable System Variables	27
Table 5-3 – Optional Configurable Pathname Variables	29

Introduction

(This introduction is not a normative part of P1003.1d, Draft Standard for Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API)—Amendment x: Additional Realtime Extensions [C Language], but is included for information only.)

1 *Editor's Note: This Introduction consists of material that will eventually be integrated into the*
2 *base POSIX.1 standard's introduction (and the portion of Annex B that contains general rationale*
3 *about the standard). The Introduction contains text that was previously held in either the Fore-*
4 *word or Scope. As this portion of the standard is for information only (nonnormative), specific*
5 *details of the integration with POSIX.1 are left as an editorial exercise. The Section and Subsection*
6 *structure of this document follows that of ISO/IEC 9945-1:1996. Sections that are not amended by*
7 *this standard are omitted.*

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

10 This standard defines systems interfaces to support the source portability of appli-
11 cations with realtime requirements. The system interfaces are all extensions of or
12 additions to *Portable Operating System Interface for Computer Environments*
13 (ISO/IEC 9945-1: 1990), as amended by IEEE-1003.1b, IEEE-1003.1c, and IEEE-
14 1003.1i. Although rooted in the culture defined by ISO/IEC 9945-1: 1990, they are
15 focused upon the realtime application requirements, which were beyond its scope.
16 The interfaces included in this standard are additions to the set required to make
17 ISO/IEC 9945-1: 1990 minimally usable to realtime applications on single processor
18 systems.

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

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

22 The key elements of defining the scope are a) defining a sufficient set of functional-
23 ity to cover the realtime application program domain in the areas not covered by
24 IEEE-1003.1b, and IEEE-1003.1c; b) defining sufficient performance constraints
25 and performance-related functions to allow a realtime application to achieve
26 deterministic response from the system; and c) specifying changes or additions to
27 improve or complete the definition of the facilities specified in the previous real-
28 time or threads extensions IEEE-1003.1b, and IEEE-1003.1c.

29 Wherever possible, the requirements of other application environments were
30 included in the interface definition. The specific areas are noted in the scope over-
31 views of each of the interface areas given below.

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

- 33 • **Spawn a Process:** new system services to spawn the execution of a new pro-
34 cess in an efficient manner.
- 35 • **Timeouts for some blocking services:** additional services that provide a
36 timeout capability to system services already defined in POSIX.1b and

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

- 37 POSIX.1c, thus allowing the application to include better error detection
38 and recovery capabilities.
- 39 • **Sporadic Server Scheduling:** the addition of a new scheduling policy
40 appropriate for scheduling aperiodic processes or threads in hard realtime
41 applications.
 - 42 • **Execution Time Clocks and Timers:** the addition of new clocks that meas-
43 ure the execution times of processes or threads, and the possibility to create
44 timers based upon these clocks, for runtime detection (and treatment) of
45 execution time overruns.
 - 46 • **Advisory Information for File Management:** addition of services that allow
47 the application to specify advisory information that can be used by the sys-
48 tem to achieve better or even deterministic response times in file manage-
49 ment or input&output operations.

50 There are two other functional areas that were included in the scope of this stan- C
51 dard, but the Ballot Group considered that they were not ready yet for standardi- C
52 zation: C

- 53 • **Device Control:** a new service to pass control information and commands
54 between the application and device drivers.
- 55 • **Interrupt Control:** new services that allow the application to directly han-
56 dle hardware interrupts.

57 This standard has been defined exclusively at the source code level, for the C pro-
58 gramming language. Although the interfaces will be portable, some of the param-
59 eters used by an implementation may have hardware or configuration dependen-
60 cies.

61 **Related Standards Activities**

62 Activities to extend this standard to address additional requirements are in pro-
63 gress, and similar efforts can be anticipated in the future.

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

- 66 (1) Additional System Application Program Interfaces in C Language
- 67 (2) Ada, and FORTRAN language bindings to (1)
- 68 (3) Shell and utility facilities
- 69 (4) Verification testing methods

70 _____
71 2) A *Standards Status Report* that lists all current IEEE Computer Society standards projects is
72 available from the IEEE Computer Society, 1730 Massachusetts Avenue NW, Washington, DC
73 20036-1903; Telephone: +1 202 371-0101; FAX: +1 202 728-9614. Working drafts of POSIX
74 standards under development are also available from this office.

- 75 (5) Realtime facilities
- 76 (6) Tracing facilities
- 77 (7) Fault tolerance
- 78 (8) Checkpoint/restart facilities
- 79 (9) Resource limiting facilities
- 80 (10) Network interface facilities
- 81 (11) System administration
- 82 (12) Profiles describing application- or user-specific combinations of Open Sys-
83 tems standards
- 84 (13) An overall guide to POSIX-based or related Open Systems standards and
85 profiles

86 Extensions are approved as “amendments” or “revisions” to this document, follow-
87 ing the IEEE and ISO/IEC Procedures.

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

90 If you have interest in participating in the PASC working groups addressing these
91 issues, please send your name, address, and phone number to the:

92 Secretary, IEEE Standards Board,
93 Institute of Electrical and Electronics Engineers, Inc.,
94 P.O. Box 1331,
95 445 Hoes Lane,
96 Piscataway, NJ 08855-1331,

97 and ask to have this forwarded to the chairperson of the appropriate PASC work-
98 ing group. If you have interest in participating in this work at the international
99 level, contact your ISO/IEC national body.

100 P1003.1d was prepared by the System Services Working Group—Realtime, spon-
101 sored by the Portable Application Standards Committee of the IEEE Computer
102 Society. At the time this standard was approved, the membership of the System
103 Services Working Group—Realtime was as follows:

104 **Portable Application Standards Committee**

105 Chair: Lowell Johnson
106 Vice Chair: Joe Gwinn
107 Treasurer: Curtis Royster
108 Secretary: Nick Stoughton

109 **System Services Working Group**

110 Chair: Jason Zions
111 Vice Chair: Joe Gwinn

112 **System Services Working Group—Realtime**

113 Chair: Joe Gwinn
114 Bill Corwin (until 1995)
115 Editor: Michael González
116 Bob Luken (until 1997)
117 Secretary: Karen Gordon
118 Lee Schemerhorn (until 1995)

119 **Ballot Coordinator**

120 Jim Oblinger
121 Duane Hughes (until 1996)

122 **Technical Reviewers**

123 Frank Prindle Michael González Karen Gordon
124 Joe Gwinn Peter Dibble Steve Brosky
125 Duane Hughes

126 **Working Group**

127 to be supplied to be supplied to be supplied

128 The following persons were members of the 1003.1d Balloting Group that
129 approved the standard for submission to the IEEE Standards Board:

130 Institutional Representatives <To be filled in>

131 Individual Balloters <To be filled in>

132 When the IEEE Standards Board approved this standard on *<date to be pro-*
133 *vided>*, it had the following membership:

134 (to be pasted in by IEEE)

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

Section 1: General

1 1.1 Scope

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

5 This standard will not change the base standard which it amends (including any
6 existing amendments) in such a way as to cause implementations or strictly con-
7 forming applications to no longer conform.

8 The scope is to take existing realtime operating system practice and add it to the
9 base standard. The definition of *realtime* used in defining the scope of this stan-
10 dard is:

11 “Realtime in operating systems: the ability of the operating system to provide
12 a required level of service in a bounded response time”

13 The key elements of defining the scope are:

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

49 **1.3 Conformance**50 **1.3.1 Implementation Conformance**

51 ⇒ **1.3.1.3 Conforming Implementation Options** *Add the following to the*
 52 *table of implementation options that warrant requirement by applications or in*
 53 *specifications:*

54	{_POSIX_ADVISORY_INFO}	Advisory Information option (in 2.9.3)	C
55	{_POSIX_CPUTIME}	Process CPU-Time Clocks option (in 2.9.3)	C
56	{_POSIX_SPAWN}	Spawn option (in 2.9.3)	C
57	{_POSIX_SPORADIC_SERVER}	Process Sporadic Server option (in 2.9.3)	C
58	{_POSIX_THREAD_CPUTIME}	Thread CPU-Time Clocks option (in 2.9.3)	C
59	{_POSIX_THREAD_SPORADIC_SERVER}	Thread Sporadic Server option (in 2.9.3)	C
60	{_POSIX_TIMEOUTS}	Timeouts option (in 2.9.3)	C

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 2: Terminology and General Requirements

1

D

2.2 Definitions

2.2.2 General Terms

⇒ **2.2.2 General Terms** *Modify the contents of subclause 2.2.2, General Terms, 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. Notice that 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. This need not always be the case 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 the purposes of this standard, the following abbreviations apply:

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

23 **2.2.3.2 POSIX.1:** ISO/IEC 9945-1: 1996, (IEEE Std 1003.1-1996), *Information*
 24 *Technology-Portable Operating System Interface (POSIX)-Part 1: System Applica-*
 25 *tion Program Interface (API) [C Language]*

26 **2.2.3.3 POSIX.1b:** IEEE Std. 1003.1b:1993, *Information Technology — Portable*
 27 *Operating System Interface (POSIX) — Part 1: System Application Program Inter-*
 28 *face (API) — Amendment b: Realtime Extensions [C Language]*, as amended by
 29 *IEEE Std. 1003.1i:1995, Information Technology — Portable Operating System*
 30 *Interface (POSIX) — Part 1: System Application Program Interface (API) —*
 31 *Amendment i: Technical Corrigenda to Realtime Extension [C Language]*.

32 **2.2.3.4 POSIX.1c:** IEEE Std. 1003.1c:1995, *Information Technology — Portable*
 33 *Operating System Interface (POSIX) — Part 1: System Application Program Inter-*
 34 *face (API) — Amendment c: Threads Extension [C Language]*

35 **2.2.3.5 POSIX.1d:** IEEE P1003.1d, *This standard.*

36 NOTE: The above reference will be changed to reflect the final POSIX.1d standard.

C

37 **2.2.3.6 POSIX.5** ISO/IEC 14519:1998 {B1}, *POSIX Ada Language Interfaces—*
 38 *Binding for System Application Program Interfaces (API) including Realtime*
 39 *Extensions.* (this standard includes IEEE Std. 1003.5-1992 and IEEE Std.
 40 1003.5b-1996).

C

C

C

C

41 **2.3 General Concepts**

42 ⇒ **2.3 General Concepts — measurement of execution time:** *Add the fol-*
 43 *lowing subclause, in the proper order, to the existing General Concept items:*

44 **2.3.1 measurement of execution time:**

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

C

49

C

50 2.7 C Language Definitions

51 C

52 2.7.3 Headers and Function Prototypes

53 ⇒ **2.7.3 Headers and Function Prototypes** *Add the following text after the* C
 54 *sentence “For other functions in this part of ISO/IEC 9945, the prototypes or* C
 55 *declarations shall appear in the headers listed below.”:* C

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

59 ⇒ **2.7.3 Headers and Function Prototypes** *Modify the contents of subclause* C
 60 *2.7.3 to add the following optional headers and functions, at the end of the* C
 61 *current list of headers and functions.* C

62 If the Advisory Information option is supported: C

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

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

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

66 If the Thread CPU-Time Clocks option is supported: C

67 <pthread.h> *pthread_getcpuclockid()* C

68 If the Threads option and the Timeouts option are supported: C

69 <pthread.h> *pthread_mutex_timedlock()* C

70 If the Semaphores option and the Timeouts option are supported: C

71 <semaphore.h> *sem_timedwait()*

72 If the Spawn option is supported: C

73 <spawn.h> *posix_spawn(), posix_spawnnp(),* C

74 *posix_spawn_file_actions_init(),* C

75 *posix_spawn_file_actions_destroy(),* C

76 *posix_spawn_file_actions_addclose(),* C

77 *posix_spawn_file_actions_adddup2(),* C

78 *posix_spawn_file_actions_addopen(),* C

79 *posix_spawnattr_init(), posix_spawnattr_destroy(),* D

80 *posix_spawnattr_getflags(), posix_spawnattr_setflags(),* D

81 *posix_spawnattr_getpgroup(),* D

82 *posix_spawnattr_setpgroup(),* D

83 *posix_spawnattr_getsigmask(),* D

84 *posix_spawnattr_setsigmask(),* D

85 *posix_spawnattr_getsigdefault(),* D

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

86		<i>posix_spawnattr_setsigdefault()</i>	D
87	If the Spawn option and the Process Scheduling option are supported:		D
88	<spawn.h>	<i>posix_spawnattr_getschedpolicy()</i> ,	D
89		<i>posix_spawnattr_setschedpolicy()</i> ,	D
90		<i>posix_spawnattr_getschedparam()</i> ,	D
91		<i>posix_spawnattr_setschedparam()</i>	D
92	If the Advisory Information option is supported:		
93	<stdlib.h>	<i>posix_memalign()</i>	
94	If the Process CPU-Time Clocks option is supported:		C
95	<time.h>	<i>clock_getcpuclockid()</i>	

96 2.8 Numerical Limits

97 2.8.2 Minimum Values

- 98 ⇒ **2.8.2 Minimum Values** *Add the following text after the sentence starting* C
 99 *“The symbols in Table 2-3 shall be defined in...”* C
- 100 The symbols in Table 2-4 shall be defined in <limits.h> with the values C
 101 shown if the associated option is supported. C
- 102 ⇒ **2.8.2 Minimum Values** *Add Table 2-4, described below, after Table 2-3 and* C
 103 *renumber other tables in this section accordingly.* C

104 **Table 2-4 – Optional Minimum Values** C

105 Name	Description	Value	Option
106 {_POSIX_SS_REPL_MAX}	107 The number of replenish- 108 ment operations that may be 109 simultaneously pending for 110 a particular sporadic server scheduler.	4	Process Sporadic Server or Thread Sporadic Server

111 2.8.4 Run-Time Invariant Values (Possibly Indeterminate)

112 D

113 ⇒ **2.8.4 Run-Time Invariant Values (Possibly Indeterminate)** *Replace the* C
 114 *whole subclause by the following text:* C

115 The symbols that appear in Table 2-5 that have determinate values shall be C
 116 defined in `<limits.h>`. The symbols that appear in Table 2-6 that have deter- C
 117 minate values shall be defined in `<limits.h>` if the associated option is sup- C
 118 ported. If any of the values in Table 2-5 or Table 2-6 have a value that is C
 119 greater than or equal to the stated minimum, but is indeterminate, a definition C
 120 for that value shall not be defined in `<limits.h>`. C

121 This might depend on the amount of available memory space on a specific C
 122 instance of a specific implementation. For the values defined in Table 2-5, the C
 123 actual value supported by a specific instance shall be provided by the `sysconf()` C
 124 function. For the values defined in Table 2-6, the actual value supported by a C
 125 specific instance shall be provided by the `sysconf()` function if the associated C
 126 option is supported. C

127 ⇒ **2.8.4 Run-Time Invariant Values (Possibly Indeterminate)** *Add* C
 128 *Table 2-6, described next, after Table 2-5, and renumber other tables in this* C
 129 *Section accordingly.* C

130 **Table 2-6 – Optional Run-Time Invariant Values (Possibly Indeterm.)** C

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

138 2.8.5 Pathname Variable Values

139 ⇒ **2.8.5 Pathname Variable Values** *Replace the reference to Table 2-6 in the* C
 140 *first paragraph of this subclause by:* C

141 Table 2-6 or Table 2-7 C

142 ⇒ **2.8.5 Pathname Variable Values** *Replace the sentence “The actual value* D
 143 *supported for a specific pathname shall be provided by the `pathconf()` function”* C
 144 *with the following text:* C

145 For the values defined in Table 2-6, the actual value supported for a specific C
 146 pathname shall be provided by the `pathconf()` function. For the values defined C
 147 in Table 2-7, the actual value supported for a specific pathname shall be pro- C
 148 vided by the `pathconf()` function if the associated option is supported. C

149 ⇒ **2.8.5 Pathname Variable Values** *Add Table 2-7, described next, after Table* C
 150 *2-6, and renumber other tables in this Section accordingly:* C

151 **Table 2-7 – Optional Pathname Variable Values** C

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.	<i>not specified</i>	Advisory Information
{POSIX_ALLOC_SIZE_MIN}	Minimum number of bytes of storage actually allocated for any portion of a file.	<i>not specified</i>	Advisory Information
{POSIX_REC_MAX_XFER_SIZE}	Maximum recommended file transfer size.	<i>not specified</i>	Advisory Information
{POSIX_REC_MIN_XFER_SIZE}	Minimum recommended file transfer size.	<i>not specified</i>	Advisory Information
{POSIX_REC_XFER_ALIGN}	Recommended file transfer buffer alignment.	<i>not specified</i>	Advisory Information

170 2.9 Symbolic Constants

171 2.9.3 Compile-Time Symbolic Constants for Portability Specifications

172 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** C
 173 *Change the first words in the first paragraph, currently saying “The constants* C
 174 *in Table 2-10 may be used...” to the following:* C

175 The constants in Table 2-10 and Table 2-11 may be used... C

176 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** C
 177 *Add the following sentence at the end of the first paragraph:* C

178 If any of the constants in Table 2-11 is defined, it shall be defined with the C
 179 value shown in that Table. This value represents the version of the associated C
 180 option that is supported by the implementation. C

181 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications** C
 182 *Add Table 2-11, shown below, after Table 2-10 renumbering all subsequent* C
 183 *tables accordingly.* C

184 **Table 2-11 – Versioned Compile-Time Symbolic Constants** C

Name	Value	Description
{_POSIX_ADVISORY_INFO}	199ymmL	If this symbol is defined, the implementation supports the Advisory Information option.
{_POSIX_CPUTIME}	199ymmL	If this symbol is defined, the implementation supports the Process CPU-Time Clocks option.
{_POSIX_SPAWN}	199ymmL	If this symbol is defined, the implementation supports the Spawn option.
{_POSIX_SPORADIC_SERVER}	199ymmL	If this symbol is defined, the implementation supports the Process Sporadic Server option.
{_POSIX_THREAD_CPUTIME}	199ymmL	If this symbol is defined, the implementation supports the Thread CPU-Time Clocks option.
{_POSIX_THREAD_SPORADIC_SERVER}	199ymmL	If this symbol is defined, the implementation supports the Thread Sporadic Server option.
{_POSIX_TIMEOUTS}	199ymmL	If this symbol is defined, the implementation supports the Timeouts option.

207 NOTE: (Editor's note) The value 199ymmL corresponds to the date of approval of POSIX.1d. C

208 C

209 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**
 210 *Add the following paragraphs:*

211 If the symbol {_POSIX_SPORADIC_SERVER} is defined, then the symbol
 212 {_POSIX_PRIORITY_SCHEDULING} shall also be defined. If the symbol
 213 {_POSIX_THREAD_SPORADIC_SERVER} is defined, then the symbol {_POSIX_
 214 THREAD_PRIORITY_SCHEDULING} shall also be defined.

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 3: Process Management

1 **3.1 Process Creation and Execution**

2 **3.1.1 Process Creation**

3 **3.1.1.2 Description**

4 ⇒ **3.1.1.2 Process Creation — Description** *Add the following paragraphs to*
 5 *the description of the fork() function:*

6 If `{_POSIX_CPUTIME}` is defined: E

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

9 If `{_POSIX_THREAD_CPUTIME}` is defined: E

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

12 **3.1.2 Execute a File**

13 **3.1.2.2 Description**

14 ⇒ **3.1.2.2 Execute a File — Description** *Add the following paragraph to the*
 15 *description of the family of exec functions.*

16 If `{_POSIX_CPUTIME}` is defined: E

17 The new process image shall inherit the CPU-time clock of the calling
 18 process image. This means that the process CPU-time clock of the pro-
 19 cess being *execed* shall not be reinitialized or altered as a result of the
 20 *exec* function other than to reflect the time spent by the process execut-
 21 ing the *exec* function itself. C

22 If `{_POSIX_THREAD_CPUTIME}` is defined: E

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

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

25 ⇒ **3.1 Process Creation** *Add the following subclauses:*

26 **3.1.4 Spawn File Actions** C

27 Functions: *posix_spawn_file_actions_init()*, *posix_spawn_file_actions_destroy()*, C
 28 *posix_spawn_file_actions_addclose()*, *posix_spawn_file_actions_adddup2()*, C
 29 *posix_spawn_file_actions_addopen()*. C

30 **3.1.4.1 Synopsis** C

31 #include <sys/types.h> C
 32 #include <spawn.h> C

33 int posix_spawn_file_actions_init(C
 34 posix_spawn_file_actions_t *file_actions); C

35 int posix_spawn_file_actions_destroy(C
 36 posix_spawn_file_actions_t *file_actions); C

37 int posix_spawn_file_actions_addclose(C
 38 posix_spawn_file_actions_t *file_actions, C
 39 int fildes); C

40 int posix_spawn_file_actions_adddup2(C
 41 posix_spawn_file_actions_t *file_actions, C
 42 int fildes, int newfildes); C

43 int posix_spawn_file_actions_addopen(C
 44 posix_spawn_file_actions_t *file_actions, C
 45 int fildes, const char *path, C
 46 int oflag, mode_t mode); C

47 **3.1.4.2 Description** C

48 If `{_POSIX_SPAWN}` is defined: E

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

55 The *posix_spawn_file_actions_init()* function initializes the object refer- C
 56 enced by *file_actions* to contain no file actions for *posix_spawn()* or C
 57 *posix_spawnp()* to perform. C

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

58	The effect of initializing an already initialized spawn file actions object is	C
59	undefined.	C
60	The <i>posix_spawn_file_actions_destroy()</i> function destroys the object refer-	C
61	enced by <i>file_actions</i> ; the object becomes, in effect, uninitialized. An imple-	C
62	mentation may cause <i>posix_spawn_file_actions_destroy()</i> to set the object	C
63	referenced by <i>file_actions</i> to an invalid value. A destroyed spawn file actions	C
64	object can be reinitialized using <i>posix_spawn_file_actions_init()</i> ; the results	C
65	of otherwise referencing the object after it has been destroyed are	C
66	undefined.	C
67	The <i>posix_spawn_file_actions_addclose()</i> function adds a close action to the	C
68	object referenced by <i>file_actions</i> that will cause the file descriptor <i>fil-des</i> to	C
69	be closed (as if <i>close(fil-des)</i> had been called) when a new process is spawned	C
70	using this file actions object.	C
71	The <i>posix_spawn_file_actions_adddup2()</i> function adds a dup2 action to the	C
72	object referenced by <i>file_actions</i> that will cause the file descriptor <i>fil-des</i> to	C
73	be duplicated as <i>newfil-des</i> (as if <i>dup2(fil-des, newfil-des)</i> had been called)	C
74	when a new process is spawned using this file actions object.	C
75	The <i>posix_spawn_file_actions_addopen()</i> function adds an open action to	C
76	the object referenced by <i>file_actions</i> that will cause the file named by <i>path</i>	C
77	to be opened (as if <i>open(path, oflag, mode)</i> had been called, and the returned	C
78	file descriptor, if not <i>fil-des</i> , had been changed to <i>fil-des</i>) when a new process	C
79	is spawned using this file actions object. If <i>fil-des</i> was already an open file	C
80	descriptor, it shall be closed before the new file is opened.	C
81	A spawn file actions object, when passed to <i>posix_spawn()</i> or	D
82	<i>posix_spawnnp()</i> , shall specify how the set of open file descriptors in the cal-	D
83	ling process is transformed into a set of potentially open file descriptors for	D
84	the spawned process. This transformation shall be as if the specified	D
85	sequence of actions was performed exactly once, in the context of the	D
86	spawned process (prior to execution of the new process image), in the order	D
87	in which the actions were added to the object; additionally, when the new	D
88	process image is executed, any file descriptor (from this new set) which has	D
89	its FD_CLOEXEC flag set will be closed (see 3.1.6).	D
90	Otherwise :	C
91	Either the implementation shall support the	C
92	<i>posix_spawn_file_actions_init()</i> , <i>posix_spawn_file_actions_destroy()</i> ,	C
93	<i>posix_spawn_file_actions_addclose()</i> , <i>posix_spawn_file_actions_adddup2()</i> ,	C
94	and <i>posix_spawn_file_actions_addopen()</i> functions as described above, or	C
95	these functions shall not be provided.	C
96	3.1.4.3 Returns	C
97	Upon successful completion, the <i>posix_spawn_file_actions_init()</i> ,	C
98	<i>posix_spawn_file_actions_destroy()</i> , <i>posix_spawn_file_actions_addclose()</i> ,	C
99	<i>posix_spawn_file_actions_adddup2()</i> , or <i>posix_spawn_file_actions_addopen()</i>	C
100	operation shall return zero. Otherwise an error number shall be returned to	C

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

101	indicate the error.	C
102	3.1.4.4 Errors	C
103	For each of the following conditions, if the condition is detected, the	C
104	<i>posix_spawn_file_actions_init()</i> , <i>posix_spawn_file_actions_addclose()</i> ,	C
105	<i>posix_spawn_file_actions_adddup2()</i> , or <i>posix_spawn_file_actions_addopen()</i> func-	C
106	tion shall return the corresponding error number:	C
107	[ENOMEM] Insufficient memory exists to initialize or add to the spawn file	C
108	actions object.	C
109	For each of the following conditions, if the condition is detected, the	C
110	<i>posix_spawn_file_actions_destroy()</i> , <i>posix_spawn_file_actions_addclose()</i> ,	C
111	<i>posix_spawn_file_actions_adddup2()</i> , or <i>posix_spawn_file_actions_addopen()</i> func-	C
112	tion shall return the corresponding error number:	C
113	[EINVAL] The value specified by <i>file_actions</i> is invalid.	C
114	For each of the following conditions, the <i>posix_spawn_file_actions_addclose()</i> ,	D
115	<i>posix_spawn_file_actions_adddup2()</i> , or <i>posix_spawn_file_actions_addopen()</i> func-	D
116	tion shall return the corresponding error number:	D
117	[EBADF] The value specified by <i>fdes</i> is negative or greater than or equal	D
118	to {OPEN_MAX}.	D
119	It shall not be considered an error for the <i>fdes</i> argument passed to the	D
120	<i>posix_spawn_file_actions_addclose()</i> , <i>posix_spawn_file_actions_adddup2()</i> , or	D
121	<i>posix_spawn_file_actions_addopen()</i> functions to specify a file descriptor for which	D
122	the specified operation could not be performed at the time of the call. Any such	D
123	error will be detected when the associated file actions object is later used during a	D
124	<i>posix_spawn()</i> or <i>posix_spawnnp()</i> operation.	D
125	3.1.4.5 Cross-References	C
126	<i>close()</i> , 6.3.1; <i>dup2()</i> , 6.2.1; <i>open()</i> , 5.3.1; <i>posix_spawn()</i> , 3.1.6; <i>posix_spawnnp()</i> ,	C
127	3.1.6;	C
128	3.1.5 Spawn Attributes	D
129	Functions: <i>posix_spawnattr_init()</i> , <i>posix_spawnattr_destroy()</i> ,	D
130	<i>posix_spawnattr_getflags()</i> , <i>posix_spawnattr_setflags()</i> ,	D
131	<i>posix_spawnattr_getpgroup()</i> , <i>posix_spawnattr_setpgroup()</i> ,	D
132	<i>posix_spawnattr_getschedpolicy()</i> , <i>posix_spawnattr_setschedpolicy()</i> ,	D
133	<i>posix_spawnattr_getschedparam()</i> , <i>posix_spawnattr_setschedparam()</i> ,	D
134	<i>posix_spawnattr_getsigmask()</i> , <i>posix_spawnattr_setsigmask()</i> ,	D
135	<i>posix_spawnattr_getsigdefault()</i> , <i>posix_spawnattr_setsigdefault()</i> .	D

136	3.1.5.1 Synopsis	D
137	#include <sys/types.h>	D
138	#include <signal.h>	D
139	#include <spawn.h>	D
140	int posix_spawnattr_init (posix_spawnattr_t *attr);	D
141	int posix_spawnattr_destroy (posix_spawnattr_t *attr);	D
142	int posix_spawnattr_getflags (const posix_spawnattr_t *attr,	D
143	short *flags);	D
144	int posix_spawnattr_setflags (posix_spawnattr_t *attr,	D
145	short flags);	D
146	int posix_spawnattr_getpgroup (const posix_spawnattr_t *attr,	D
147	pid_t *pgroup);	D
148	int posix_spawnattr_setpgroup (posix_spawnattr_t *attr,	D
149	pid_t pgroup);	D
150	int posix_spawnattr_getsigmask (const posix_spawnattr_t *attr,	D
151	sigset_t *sigmask);	D
152	int posix_spawnattr_setsigmask (posix_spawnattr_t *attr,	D
153	const sigset_t *sigmask);	D
154	int posix_spawnattr_getdefault (const posix_spawnattr_t *attr,	D
155	sigset_t *sigdefault);	D
156	int posix_spawnattr_setdefault (posix_spawnattr_t *attr,	D
157	const sigset_t *sigdefault);	D
158	#include <sched.h>	D
159	int posix_spawnattr_getschedpolicy (const posix_spawnattr_t *attr,	D
160	int *schedpolicy);	D
161	int posix_spawnattr_setschedpolicy (posix_spawnattr_t *attr,	D
162	int schedpolicy);	D
163	int posix_spawnattr_getschedparam (const posix_spawnattr_t *attr,	D
164	struct sched_param *schedparam);	D
165	int posix_spawnattr_setschedparam (posix_spawnattr_t *attr,	D
166	const struct sched_param *schedparam);	D
167	3.1.5.2 Description	D
168	If <code>{_POSIX_SPAWN}</code> is defined:	E
169	A spawn attributes object is of type <i>posix_spawnattr_t</i> (defined in	D
170	<spawn.h>) and is used to specify the inheritance of process attributes	D
171	across a spawn operation. This standard does not define comparison or	E
172	assignment operators for the type <i>posix_spawnattr_t</i> .	D
173	The function <i>posix_spawnattr_init()</i> initializes a spawn attributes object	D
174	<i>attr</i> with the default value for all of the individual attributes used by the	D

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

175 implementation. D

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

178 The resulting spawn attributes object (possibly modified by setting indivi- D
 179 dual attribute values), is used to modify the behavior of *posix_spawn()* or D
 180 *posix_spawnp()* (see 3.1.6). After a spawn attributes object has been used to D
 181 spawn a process by a call to a *posix_spawn()* or *posix_spawnp()*, any func- D
 182 tion affecting the attributes object (including destruction) does not affect D
 183 any process that has been spawned in this way. D

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

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

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

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

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

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

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

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

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

226 Otherwise: D

227 Either the implementation shall support the `posix_spawnattr_init()`, D
 228 `posix_spawnattr_destroy()`, `posix_spawnattr_getflags()`, D
 229 `posix_spawnattr_setflags()`, `posix_spawnattr_getpgroup()`, D
 230 `posix_spawnattr_setpgroup()`, `posix_spawnattr_getsigmask()`, D
 231 `posix_spawnattr_setsigmask()`, `posix_spawnattr_getsigdefault()`, and D
 232 `posix_spawnattr_setsigdefault()` functions as described above or these func- D
 233 tions shall not be provided. D

234 If `{_POSIX_SPAWN}` and `{_POSIX_PRIORITY_SCHEDULING}` are both defined: E

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

239 The `posix_spawnattr_setschedpolicy()` function is used to set the `spawn-` D
 240 `schedpolicy` attribute in an initialized attributes object referenced by D
 241 `attr`. The `posix_spawnattr_getschedpolicy()` function obtains the value of the D
 242 `spawn-schedpolicy` attribute from the attributes object referenced by D
 243 `attr`. D

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

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

253 Otherwise: D

254 Either the implementation shall support the D
 255 `posix_spawnattr_getschedpolicy()`, `posix_spawnattr_setschedpolicy()`, D
 256 `posix_spawnattr_getschedparam()`, and `posix_spawnattr_setschedparam()` D
 257 functions as described above or these functions shall not be provided. D

258 Additional attributes, their default values, and the names of the associated func- D
 259 tions to get and set those attribute values are implementation defined. D

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

260	3.1.5.3 Returns	D
261	If successful, the <i>posix_spawnattr_init()</i> , <i>posix_spawnattr_destroy()</i> ,	D
262	<i>posix_spawnattr_setflags()</i> , <i>posix_spawnattr_setpgroup()</i> ,	D
263	<i>posix_spawnattr_setschedpolicy()</i> , <i>posix_spawnattr_setschedparam()</i> ,	D
264	<i>posix_spawnattr_setsigmask()</i> , and <i>posix_spawnattr_setsigdefault()</i> functions	D
265	shall return zero. Otherwise, an error number shall be returned to indicate the	D
266	error.	D
267	If successful, the <i>posix_spawnattr_getflags()</i> , <i>posix_spawnattr_getpgroup()</i> ,	D
268	<i>posix_spawnattr_getschedpolicy()</i> , <i>posix_spawnattr_getschedparam()</i> ,	D
269	<i>posix_spawnattr_getsigmask()</i> , and <i>posix_spawnattr_getsigdefault()</i> functions	D
270	shall return zero and respectively store the value of the spawn-flags, spawn-	D
271	pgroup, spawn-schedpolicy, spawn-schedparam, spawn-sigmask, or	D
272	spawn-sigdefault, attribute of <i>attr</i> into the object referenced by the <i>flags</i> ,	D
273	<i>pgroup</i> , <i>schedpolicy</i> , <i>schedparam</i> , <i>sigmask</i> or <i>sigdefault</i> parameter, respectively.	D
274	Otherwise, an error number shall be returned to indicate the error.	D
275	3.1.5.4 Errors	D
276	If any of the following conditions occur, the <i>posix_spawnattr_init()</i> function shall	D
277	return the corresponding error value:	D
278	[ENOMEM] Insufficient memory exists to initialize the spawn attributes	D
279	object.	D
280	For each of the following conditions, if the condition is detected, the	D
281	<i>posix_spawnattr_destroy()</i> , <i>posix_spawnattr_getflags()</i> ,	D
282	<i>posix_spawnattr_setflags()</i> , <i>posix_spawnattr_getpgroup()</i> ,	D
283	<i>posix_spawnattr_setpgroup()</i> , <i>posix_spawnattr_getschedpolicy()</i> ,	D
284	<i>posix_spawnattr_setschedpolicy()</i> , <i>posix_spawnattr_getschedparam()</i> ,	D
285	<i>posix_spawnattr_setschedparam()</i> , <i>posix_spawnattr_getsigmask()</i> ,	D
286	<i>posix_spawnattr_setsigmask()</i> , <i>posix_spawnattr_getsigdefault()</i> , and	D
287	<i>posix_spawnattr_setsigdefault()</i> functions shall return the corresponding error	D
288	value:	D
289	[EINVAL] The value specified by <i>attr</i> is invalid.	D
290	For each of the following conditions, if the condition is detected, the	D
291	<i>posix_spawnattr_setflags()</i> , <i>posix_spawnattr_setpgroup()</i> ,	D
292	<i>posix_spawnattr_setschedpolicy()</i> , <i>posix_spawnattr_setschedparam()</i> ,	D
293	<i>posix_spawnattr_setsigmask()</i> , and <i>posix_spawnattr_setsigdefault()</i> functions	D
294	shall return the corresponding error value:	D
295	[EINVAL] The value of the attribute being set is not valid.	D
296	3.1.5.5 Cross-References	D
297	<i>posix_spawn()</i> , 3.1.6; <i>posix_spawnp()</i> , 3.1.6.	D

298 **3.1.6 Spawn a Process**299 Functions: *posix_spawn()*, *posix_spawnp()*.300 **3.1.6.1 Synopsis**

```

301 #include <sys/types.h> C
302 #include <spawn.h>
303 int posix_spawn( pid_t *pid,
304                 const char *path,
305                 const posix_spawn_file_actions_t *file_actions, C
306                 const posix_spawnattr_t *attrp, D
307                 char * const argv[],
308                 char * const envp[] );
309 int posix_spawnp( pid_t *pid,
310                  const char *file,
311                  const posix_spawn_file_actions_t *file_actions, C
312                  const posix_spawnattr_t *attrp, D
313                  char * const argv[],
314                  char * const envp[] );

```

315 **3.1.6.2 Description**316 If `{_POSIX_SPAWN}` is defined: E

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

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

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

323 Where *argc* is the argument count and *argv* is an array of character
 324 pointers to the arguments themselves. In addition, the following variable:

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

326 is initialized as a pointer to an array of character pointers to the environ-
 327 ment strings.

328 C

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

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

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

338	The number of bytes available for the child process's combined argument	
339	and environment lists is {ARG_MAX}. The implementation shall specify in	
340	the system documentation (see 1.3.1) whether any list overhead, such as	D
341	length words, null terminators, pointers, or alignment bytes, is included in	
342	this total.	
343	The <i>path</i> argument to <i>posix_spawn()</i> is a pathname that identifies the new	
344	process image file to execute.	
345	The <i>file</i> parameter to <i>posix_spawnp()</i> shall be used to construct a pathname	
346	that identifies the new process image file. If the <i>file</i> parameter contains a	
347	slash character, the <i>file</i> parameter shall be used as the pathname for the	
348	new process image file. Otherwise, the path prefix for this file shall be	
349	obtained by a search of the directories passed as the environment variable	
350	PATH (see 2.6). If this environment variable is not defined, the results of	D
351	the search are implementation defined.	
352		C
353	If <i>file_actions</i> is a NULL pointer, then file descriptors open in the calling	C
354	process shall remain open in the child process, except for those whose	C
355	close-on-exec flag FD_CLOEXEC is set (see 6.5.2 and 6.5.1). For those file	C
356	descriptors that remain open, all attributes of the corresponding open file	C
357	descriptions, including file locks (see 6.5.2), shall remain unchanged.	C
358	If <i>file_actions</i> is not NULL , then the file descriptors open in the child pro-	C
359	cess shall be those open in the calling process process as modified by the	C
360	spawn file actions object pointed to by <i>file_actions</i> and the FD_CLOEXEC	C
361	flag of each remaining open file descriptor after the spawn file actions have	C
362	been processed. The effective order of processing the spawn file actions	C
363	shall be:	C
364	1. The set of open file descriptors for the child process shall initially be	C
365	the same set as is open for the calling process. All attributes of the	C
366	corresponding open file descriptions, including file locks (see 6.5.2),	C
367	shall remain unchanged.	C
368	2. The signal mask and the effective user and group IDs for the child pro-	C
369	cess shall be changed as specified in the attributes object referenced	D
370	by <i>attrp</i> .	D
371	3. The file actions specified by the spawn file actions object shall be per-	C
372	formed in the order in which they were added to the spawn file actions	C
373	object.	C
374	4. Any file descriptor which has its FD_CLOEXEC flag set (see 6.5.2) shall	C
375	be closed.	C
376	The <i>posix_spawnattr_t</i> spawn attributes object type is defined in	D
377	<spawn.h>. It shall contain at least the attributes described in 3.1.5.	D
378	If the POSIX_SPAWN_SETPGROUP flag is set in the <i>spawn-flags</i> attribute	D
379	of the object referenced by <i>attrp</i> , and the <i>spawn-pgroup</i> attribute of the	D
380	same object is non zero, then the child's process group shall be as specified	D
381	in the <i>spawn-pgroup</i> attribute of the object referenced by <i>attrp</i> .	D

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

382 As a special case, if the `POSIX_SPAWN_SETPGROUP` flag is set in the D
383 `spawn-flags` attribute of the object referenced by *attrp*, and the `spawn-` D
384 `pgroup` attribute of the same object is set to 0, then the child shall be in a D
385 new process group with a process group ID equal to its process ID. D

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

389 If `{_POSIX_PRIORITY_SCHEDULING}` is defined, and the E
390 `POSIX_SPAWN_SETSCHEDPARAM` flag is set in the `spawn-flags` attribute D
391 of the object referenced by *attrp*, but `POSIX_SPAWN_SETSCHEDULER` is not D
392 set, the new process image shall initially have the scheduling policy of the D
393 calling process with the scheduling parameters specified in the `spawn-` D
394 `schedparam` attribute of the object referenced by *attrp*. D

395 If `{_POSIX_PRIORITY_SCHEDULING}` is defined, and the E
396 `POSIX_SPAWN_SETSCHEDULER` flag is set in `spawn-flags` attribute of D
397 the object referenced by *attrp* (regardless of the setting of the D
398 `POSIX_SPAWN_SETSCHEDPARAM` flag), the new process image shall ini- D
399 tially have the scheduling policy specified in the `spawn-schedpolicy` D
400 attribute of the object referenced by *attrp* and the scheduling parameters D
401 specified in the `spawn-schedparam` attribute of the same object. D

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

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

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

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

426 Signals set to be caught by the calling process shall be set to the default
427 action in the child process.

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

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

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

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

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

445 Otherwise :

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

449 3.1.6.3 Returns

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

457 3.1.6.4 Errors

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

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

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

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

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

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

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

494 3.1.6.5 Cross-References

495 `alarm()`, 3.4.1; `chmod()`, 5.6.4; `close()`, 6.3.1; `dup2()`, 6.2.1; `exec`, 3.1.2; `_exit()`, 3.2.2; C
 496 `fcntl()`, 6.5.2; `fork()`, 3.1.1; `kill()`, 3.3.2; `open()`, 5.3.1; C
 497 `posix_spawn_file_actions_init()`, 3.1.4; `posix_spawn_file_actions_destroy()`, 3.1.4; C
 498 `posix_spawn_file_actions_addclose()`, 3.1.4; `posix_spawn_file_actions_adddup2()`, C
 499 3.1.4; `posix_spawn_file_actions_addopen()`, 3.1.4; `posix_spawnattr_init()`, 3.1.5; D
 500 `posix_spawnattr_destroy()`, 3.1.5; `posix_spawnattr_getflags()`, 3.1.5; D
 501 `posix_spawnattr_setflags()`, 3.1.5; `posix_spawnattr_getpgroup()`, 3.1.5; D
 502 `posix_spawnattr_setpgroup()`, 3.1.5; `posix_spawnattr_getschedpolicy()`, 3.1.5; D
 503 `posix_spawnattr_setschedpolicy()`, 3.1.5; `posix_spawnattr_getschedparam()`, 3.1.5; D
 504 `posix_spawnattr_setschedparam()`, 3.1.5; `posix_spawnattr_getsigmask()`, 3.1.5; D
 505 `posix_spawnattr_setsigmask()`, 3.1.5; `posix_spawnattr_getsigdefault()`, 3.1.5; D
 506 `posix_spawnattr_setsigdefault()`, 3.1.5; `sched_setparam()`, 13.3.1; D
 507 `sched_setscheduler()`, 13.3.3; `setpgid()`, 4.3.3; `setuid()`, 4.2.2; `stat()`, 5.6.2; `times()`, C
 508 4.5.2; `wait`, 3.2.1. C

509 **3.2 Process Termination**

510 **3.2.1 Wait for Process Termination**

511 **3.2.1.2 Wait for Process Termination — Description**

512 C

513 ⇒ **3.2.1.2 Wait for Process Termination — Description** *Add the following*
 514 *paragraphs after the definition of the WSTOPSIG(stat_val) macro:* C

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

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

Section 4: Process Environment

1 4.8 Configurable System Variables

2 4.8.1 Get Configurable System Variables

3 ⇒ **4.8.1.2 Get Configurable System Variables— Description** *Add the follow-* C
 4 *ing text after the sentence “The implementation shall support all of the vari-* C
 5 *ables listed in Table 4-2 and may support others”, in the second paragraph:* C

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

9 ⇒ **4.8.1.2 Get Configurable System Variables— Description** *In the second* C
 10 *paragraph, replace the text “The variables in Table 4-2 come from ...” by the* C
 11 *following:* C

12 “The variables in Table 4-2 and Table 4-3 come from ...” C

13 ⇒ **4.8.1.2 Get Configurable System Variables— Description** *Add the follow-* C
 14 *ing table:* C

15 **Table 4-3 – Optional Configurable System Variables** C

16	Variable	name Value	Required Option	17
18	{_POSIX_SPAWN}	_SC_SPAWN	Spawn	C
19	{_POSIX_TIMEOUTS}	_SC_TIMEOUTS	Timeouts	C
20	{_POSIX_CPUTIME}	_SC_CPUTIME	Process CPU-Time Clocks	C
21	{_POSIX_THREAD_CPUTIME}	_SC_THREAD_CPUTIME	Thread CPU-Time Clocks	C
22	{_POSIX_SPORADIC_SERVER}	_SC_SPORADIC_SERVER	Process Sporadic Server	C
23	{_POSIX_THREAD_SPORADIC_SERVER}	_SC_THREAD_SPORADIC_SERVER	Thread Sporadic Server	C
24	{_POSIX_ADVISORY_INFO}	_SC_ADVISORY_INFO	Advisory Information	C

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 5: Files and Directories

1 **5.7 Configurable Pathname Variables** C

2 **5.7.1 Get Configurable Pathname Variables** C

3 ⇒ **5.7.1.2 Get Configurable Pathname Variables— Description** *Add the fol-* C
 4 *lowing text after the sentence “The implementation shall support all of the* C
 5 *variables listed in Table 5-2 and may support others”, in the third paragraph:* C

6 Support for some pathname configuration variables is dependent on implemen- C
 7 tation options (see Table 5-3). Where an implementation option is not sup- C
 8 ported, the variable need not be supported. C

9 ⇒ **5.7.1.2 Get Configurable Pathname Variables— Description** *In the third* C
 10 *paragraph, replace the text “The variables in Table 5-2 come from ...” by the* C
 11 *following:* C

12 “The variables in Table 5-2 and Table 5-3 come from ...” C

13 ⇒ **5.7.1.2 Get Configurable Pathname Variables— Description** *Add the fol-* C
 14 *lowing table:* C

15 **Table 5-3 – Optional Configurable Pathname Variables** C

Variable	<i>name</i> Value	Required Option	C
{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	Advisory Information	C
{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	Advisory Information	C
{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	Advisory Information	C
{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	Advisory Information	C
{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	Advisory Information	C

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 6: Input and Output Primitives

1 **6.7 Asynchronous Input and Output**

2 **6.7.1 Data Definitions for Asynchronous Input and Output**

3 ⇒ **6.7.1.1 Asynchronous I/O Control Block** *Change the sentence, beginning*
 4 *with “The order of processing of requests submitted by processes whose*
 5 *schedulers ... ” to the following:*

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 11: Synchronization

1 11.2 Semaphore Functions

2 11.2.6 Lock a Semaphore

3 ⇒ **11.2.6 Lock a Semaphore** *Add the following function to the list:*

4 `sem_timedwait()`.

5 11.2.6.1 Synopsis

6 ⇒ **11.2.6.1 Lock a Semaphore — Synopsis** *Add the following #include and*
7 *prototype to the synopsis:*

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

C

11 11.2.6.2 Description

12 ⇒ **11.2.6.2 Lock a Semaphore — Description** *Add the following text to the*
13 *description:*

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

E

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

20 The timeout expires when the absolute time specified by `abs_timeout`
21 passes, as measured by the clock on which timeouts are based (that is,
22 when the value of that clock equals or exceeds `abs_timeout`), or if the
23 absolute time specified by `abs_timeout` has already been passed at the
24 time of the call. If the Timers option is supported, the timeout is based
25 on the `CLOCK_REALTIME` clock; if the Timers option is not supported,
26 the timeout is based on the system clock as returned by the `time()`

C

C

C

C

C

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

55 **11.2.7 Unlock a Semaphore**

56 ⇒ **11.2.7.2 Unlock a Semaphore — Description** *(The following change is*
 57 *made in a context where the Process Scheduling option is supported.) Change*
 58 *the sentence, beginning with “In the case of the schedulers ... ” to the follow-*
 59 *ing:*

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

65 **11.3 Mutexes**

66

C

67 **11.3.3 Locking and Unlocking a Mutex**

68 ⇒ **11.3.3 Locking and Unlocking a Mutex** *Add the following function to the*
 69 *list:*

70 `pthread_mutex_timedlock()`.

71 **11.3.3.1 Synopsis**

72 ⇒ **11.3.3.1 Locking and Unlocking a Mutex — Synopsis** *Add the following*
 73 *#include and prototype to the synopsis:*

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

C

77 **11.3.3.2 Description**

78 ⇒ **11.3.3.2 Locking and Unlocking a Mutex — Description** *Add the follow-*
 79 *ing text to the description:*

80 If `{_POSIX_THREADS}` and `{_POSIX_TIMEOUTS}` are both defined: E

81 The `pthread_mutex_timedlock()` function is called to lock the mutex C
 82 object referenced by `mutex`. If the mutex is already locked, the calling C
 83 thread blocks until the mutex becomes available as in the
 84 `pthread_mutex_lock()` function. If the mutex cannot be locked without
 85 waiting for another thread to unlock the mutex, this wait shall be ter-
 86 minated when the specified timeout expires.

87 The timeout expires when the absolute time specified by `abs_timeout` C
 88 passes, as measured by the clock on which timeouts are based (that is, C
 89 when the value of that clock equals or exceeds `abs_timeout`), or if the C
 90 absolute time specified by `abs_timeout` has already been passed at the C
 91 time of the call. If the Timers option is supported, the timeout is based C
 92 on the `CLOCK_REALTIME` clock; if the Timers option is not supported,
 93 the timeout is based on the system clock as returned by the `time()` func-
 94 tion. The resolution of the timeout is the resolution of the clock on
 95 which it is based. The `timespec` datatype is defined as a structure in the C
 96 header `<time.h>`. C

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

100 As a consequence of the priority inheritance rules (for mutexes initial-
 101 ized with the `PRIO_INHERIT` protocol), if a timed mutex wait is ter-
 102 minated because its timeout expires, the priority of the owner of the
 103 mutex will be adjusted as necessary to reflect the fact that this thread is
 104 no longer among the threads waiting for the mutex.

105 C

106 Otherwise:

107 Either the implementation shall support the `pthread_mutex_timedlock()`
 108 function as described above or the function shall not be provided.

109 **11.3.3.3 Returns**

110 ⇒ **11.3.3.3 Locking and Unlocking a Mutex — Returns** *Add the function*
 111 *pthread_mutex_timedlock() to the list of functions.*

112 **11.3.3.4 Errors**

113 ⇒ **11.3.3.4 Locking and Unlocking a Mutex — Errors** *Make the following*
 114 *changes to the discussion of error conditions:*

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

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

118 The process or thread would have blocked, and the *abs_timeout* parame-
 119 ter specified a nanoseconds field value less than zero or greater than or
 120 equal to 1000 million.

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

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

126

C

127 **11.3.3.5 Cross-References**

128 ⇒ **11.3.3.5 Locking and Unlocking a Mutex — Cross-References** *Add the*
 129 *following items to the cross-references:*

130 *time()*, 4.5.1; <time.h>, 14.1.

C

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 13: Execution Scheduling

1 This section describes the extension to the system interfaces to support the
2 sporadic server scheduling policy.

3 13.1 Scheduling Parameters

4 ⇒ **13.1 Scheduling Parameters** *Add the following paragraph:*

5 In addition, if `{_POSIX_SPORADIC_SERVER}` is defined or `{_POSIX_THREAD_-` E
6 `SPORADIC_SERVER}` is defined, the `sched_param` structure defined in E
7 `<sched.h>` shall contain the following members in addition to those specified
8 above:

9 Member 10 Type	Member 11 Name	Description
11 <i>int</i>	<i>sched_ss_low_priority</i>	Low scheduling priority for sporadic server.
12 <i>timespec</i>	<i>sched_ss_repl_period</i>	Replenishment period for sporadic server.
13 <i>timespec</i>	<i>sched_ss_init_budget</i>	Initial budget for sporadic server.
14 <i>int</i>	<i>sched_ss_max_repl</i>	Maximum pending replenishments for sporadic server.

15 13.2 Scheduling Policies

16 ⇒ **13.2 Scheduling Policies** *Add the following after the unnumbered table with* C
17 *the scheduling policies that shall be defined in <sched.h> :* C

18 If `{_POSIX_SPORADIC_SERVER}` is defined or `{_POSIX_THREAD_SPORADIC_-` E
19 `SERVER}` is defined, then the following scheduling policy is provided in E
20 `<sched.h>`: C

21 Symbol	Description
22 SCHED_SPORADIC	Sporadic server scheduling policy.

23 **13.2.3 SCHED_OTHER**

24 ⇒ **13.2.3 SCHED_OTHER** *Change the sentence, beginning with “The effect of*
 25 *scheduling threads with the . . . ” to the following:*

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

29 **13.2.4 SCHED_SPORADIC**

30 ⇒ **13.2.4 SCHED_SPORADIC**

31 *Add the following subclause to the Execution Scheduling section:*

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

35 The sporadic server policy is based primarily on two parameters: the replenish- C
 36 ment period and the available execution capacity. The replenishment period is C
 37 given by the *sched_ss_repl_period* member of the *sched_param* structure. The C
 38 available execution capacity is initialized to the value given by the
 39 *sched_ss_init_budget* member of the same parameter. The sporadic server pol-
 40 icy is identical to the SCHED_FIFO policy with some additional conditions that
 41 cause the thread’s assigned priority to be switched between the values specified
 42 by the *sched_priority* and *sched_ss_low_priority* members of the *sched_param* C
 43 structure. C

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

- 55 (1) When the thread at the head of the *sched_priority* list becomes a running
 56 thread, its execution time shall be limited to at most its available execu-
 57 tion capacity, plus the resolution of the execution time clock used for this C
 58 scheduling policy. This resolution shall be implementation defined. C
- 59 (2) Each time the thread is inserted at the tail of the list associated with
 60 *sched_priority* — because as a blocked thread it became runnable with
 61 priority *sched_priority* or because a replenishment operation was

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

- 62 performed — the time at which this operation is done is posted as the
63 *activation_time*.
- 64 (3) When the running thread with assigned priority equal to *sched_priority*
65 becomes a preempted thread, it becomes the head of the thread list for its
66 priority, and the execution time consumed is subtracted from the avail-
67 able execution capacity. If the available execution capacity would become C
68 negative by this operation, it shall be set to zero. D
- 69 (4) When the running thread with assigned priority equal to *sched_priority*
70 becomes a blocked thread, the execution time consumed is subtracted
71 from the available execution capacity, and a replenishment operation is
72 scheduled, as described below. If the available execution capacity would C
73 become negative by this operation, it shall be set to zero. D
- 74 (5) When the running thread with assigned priority equal to *sched_priority*
75 reaches the limit imposed on its execution time, it becomes the tail of the
76 thread list for *sched_ss_low_priority*, the execution time consumed is sub-
77 tracted from the available execution capacity (which becomes zero), and a
78 replenishment operation is scheduled, as described below.
- 79 (6) Each time a replenishment operation is scheduled, the amount of execu-
80 tion capacity to be replenished, *replenish_amount*, is set equal to the execu-
81 tion time consumed by the thread since the *activation_time*. The
82 replenishment is scheduled to occur at *activation_time* plus
83 *sched_ss_repl_period*. If the scheduled time obtained is before the current
84 time, the replenishment operation is carried out immediately. Notice
85 that there may be several replenishment operations pending at the same
86 time, each of which will be serviced at its respective scheduled time.
87 Notice also that with the above rules, the number of replenishment
88 operations simultaneously pending for a given thread that is scheduled
89 under the sporadic server policy shall not be greater than
90 *sched_ss_max_repl*.
- 91 (7) A replenishment operation consists of adding the corresponding
92 *replenish_amount* to the available execution capacity at the scheduled
93 time. If as a consequence of this operation the execution capacity would C
94 become larger than *sched_ss_initial_budget*, it shall be rounded down to a C
95 value equal to *sched_ss_initial_budget*. Additionally, if the thread was C
96 runnable or running, and with assigned priority equal to
97 *sched_ss_low_priority*, then it becomes the tail of the thread list for
98 *sched_priority*.

99 Execution time is defined in 2.2.2.

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

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

106 13.3 Process Scheduling Functions

107 13.3.1 Set Scheduling Parameters

108 ⇒ **13.3.1.2 Set Scheduling Parameters — Description** *Add the following*
 109 *paragraphs to the description of the function sched_setparam():*

110 If `{_POSIX_SPORADIC_SERVER}` is defined: E

111 If the scheduling policy of the target process is `SCHED_SPORADIC`, the
 112 value specified by the `sched_ss_low_priority` member of the `param` argu-
 113 ment shall be any integer within the inclusive priority range for the
 114 sporadic server policy. The `sched_ss_repl_period` and
 115 `sched_ss_init_budget` members of the `param` argument shall represent
 116 the time parameters to be used by the sporadic server scheduling policy
 117 for the target process. The `sched_ss_max_repl` member of the `param`
 118 argument shall represent the maximum number of replenishments that
 119 are allowed to be pending simultaneously for the process scheduled
 120 under this scheduling policy.

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

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

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

134 ⇒ **13.3.1.2 Set Scheduling Parameters — Description** *Add the* C
 135 *SCHED_SPORADIC policy to the last paragraph, that describes the cases in*
 136 *which the result of this function is implementation defined. The new para-*
 137 *graph shall be:*

138 If the current scheduling policy for the process specified by `pid` is not
 139 `SCHED_FIFO`, `SCHED_RR` or `SCHED_SPORADIC`, the result is implemen-
 140 tation defined; this includes the `SCHED_OTHER` policy.

141 13.3.3 Set Scheduling Policy and Scheduling Parameters

142 ⇒ **13.3.3.2 Set Scheduling Policy and Scheduling Parameters — Descrip-**
 143 **tion** *Add the following paragraphs to the description of the function*
 144 *sched_setscheduler():*

145 If `{_POSIX_SPORADIC_SERVER}` is defined: E

146 If the scheduling policy specified by *policy* is `SCHED_SPORADIC`, the
 147 value specified by the *sched_ss_low_priority* member of the *param* argu-
 148 ment shall be any integer within the inclusive priority range for the
 149 sporadic server policy. The *sched_ss_repl_period* and
 150 *sched_ss_init_budget* members of the *param* argument shall represent
 151 the time parameters used by the sporadic server scheduling policy for
 152 the target process. The *sched_ss_max_repl* member of the *param* argu-
 153 ment shall represent the maximum number of replenishments that are
 154 allowed to be pending simultaneously for the process scheduled under
 155 this scheduling policy.

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

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

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

166 13.4 Thread Scheduling

167 13.4.1 Thread Scheduling Attributes

168 ⇒ **13.4.1 Thread Scheduling Attributes** *Add the following paragraph after*
 169 *the paragraph that begins with “If the*
 170 *{_POSIX_THREAD_PRIORITY_SCHEDULING} option is defined, ...”:*

171 If `{_POSIX_THREAD_SPORADIC_SERVER}` is defined, the `schedparam` E
 172 attribute supports four new members that are used for the sporadic
 173 server scheduling policy. These members are *sched_ss_low_priority*,
 174 *sched_ss_repl_period*, *sched_ss_init_budget*, and *sched_ss_max_repl*.
 175 The meaning of these attributes is the same as in the definitions that
 176 appear under **Process Scheduling Attributes**.

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

177 **13.4.3 Scheduling Allocation Domain**

178 ⇒ **13.4.3 Scheduling Allocation Domain**

179 *Add the following paragraph after the fourth paragraph in 13.4.3:*

C

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

E

C

C

C

184 ⇒ **13.4.3 Scheduling Allocation Domain** *Change the first sentence of the*
 185 *fourth paragraph, currently reading “For application threads whose scheduling*
 186 *allocation domain size is greater than one, the rules defined for `SCHED_FIFO`*
 187 *and `SCHED_RR` in 13.2 shall be used in an implementation-defined manner.” to*
 188 *the following:*

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

193 **13.4.4 Scheduling Documentation**

194 ⇒ **13.4.4 Scheduling Documentation** *Change the sentence, beginning with “If*
 195 *`{_POSIX_PRIORITY_SCHEDULING}` is defined, then ... ” and ending with “...*
 196 *such a policy, are implementation defined.” to the following:*

197 If `{_POSIX_PRIORITY_SCHEDULING}` is defined, then any scheduling policies
 198 beyond `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`, and `SCHED_SPORADIC`, as
 199 as well as the effects of the scheduling policies indicated by these other values,
 200 and the attributes required in order to support such a policy, are implementa-
 201 tion defined.

E

202 **13.5 Thread Scheduling Functions**

203 **13.5.1 Thread Creation Scheduling Attributes**

204 ⇒ **13.5.1.2 Thread Creation Scheduling Attributes — Description** *Add the*
 205 *following paragraph to the description of functions*
 206 *pthread_attr_setschedpolicy() and pthread_attr_getschedpolicy():*

207 In addition, if `{_POSIX_THREAD_SPORADIC_SERVER}` is defined, the E
 208 value of *policy* may be `SCHED_SPORADIC`.

209 *Also, add the following sentences at the end of the paragraph that describes the*
 210 *functions pthread_attr_setschedparam() and pthread_attr_getschedparam():*

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

219 **13.5.2 Dynamic Thread Scheduling Parameters Access**

220 ⇒ **13.5.2.2 Dynamic Thread Scheduling Parameters Access — Descrip-**
 221 **tion** *Add the following paragraph to the description of the functions*
 222 *pthread_setschedparam() and pthread_getschedparam():*

223 If `{_POSIX_THREAD_SPORADIC_SERVER}` is defined, then the *policy* argument E
 224 may have the value `SCHED_SPORADIC`, with the exception for the
 225 *pthread_setschedparam()* function that if the scheduling policy was not
 226 `SCHED_SPORADIC` at the time of the call, it is implementation defined whether
 227 the function is supported; this means that the implementation need not allow
 228 the application to dynamically change the scheduling policy to
 229 `SCHED_SPORADIC`. The sporadic server scheduling policy has the associated
 230 parameters *sched_ss_low_priority*, *sched_ss_repl_period*, *sched_ss_init_budget*,
 231 *sched_priority*, and *sched_ss_max_repl*. The specified *sched_ss_repl_period*
 232 must be greater than or equal to the specified *sched_ss_init_budget* for the
 233 function to succeed; if it is not, then the function shall fail. The value of
 234 *sched_ss_max_repl* shall be within the inclusive range `[1, {SS_REPL_MAX}]` for
 235 the function to succeed; if not, the function shall fail.

236 ⇒ **13.5.2.4 Dynamic Thread Scheduling Parameters Access — Errors** *Add*
237 *the following error status value in the "if detected" section of the* C
238 *pthread_setschedparam() function:* C

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

Section 14: Clocks and Timers

1 14.2 Clock and Timer Functions

2 14.2.1 Clocks

3 14.2.1.2 Description

4 ⇒ **14.2.1.2 Clock and Timer Functions — Description** *Add the following*
 5 *paragraphs to the description of the functions `clock_settime()`, `clock_gettime()`,*
 6 *`clock_getres()`:*

7 If `{_POSIX_CPUTIME}` is defined, implementations shall support clock ID values E
 8 obtained by invoking `clock_getcpuclockid()`, which represent the CPU-time
 9 clock of a given process. Implementations shall also support the special
 10 `clockid_t` value `CLOCK_PROCESS_CPUTIME_ID`, which represents the
 11 CPU-time clock of the calling process when invoking one of the clock or timer
 12 functions. For these clock IDs, the values returned by `clock_gettime()` and
 13 specified by `clock_settime()` represent the amount of execution time of the pro-
 14 cess associated with the clock. Changing the value of a CPU-time clock via C
 15 `clock_settime()` shall have no effect on the behavior of the sporadic server C
 16 scheduling policy (see 13.2.4). C

17 If `{_POSIX_THREAD_CPUTIME}` is defined, implementations shall support clock E
 18 ID values obtained by invoking `pthread_getcpuclockid()`, which represent the
 19 CPU-time clock of a given thread. Implementations shall also support the spe- C
 20 cial `clockid_t` value `CLOCK_THREAD_CPUTIME_ID`, which represents the
 21 CPU-time clock of the calling thread when invoking one of the clock or timer
 22 functions. For these clock IDs, the values returned by `clock_gettime()` and
 23 specified by `clock_settime()` represent the amount of execution time of the
 24 thread associated with the clock. Changing the value of a CPU-time clock via C
 25 `clock_settime()` shall have no effect on the behavior of the sporadic server C
 26 scheduling policy (see 13.2.4). C
 27 C

28 **14.2.2 Create a Per-Process Timer**

29 **14.2.2.2 Description**

30 ⇒ **14.2.2.2 Create a Per-Process Timer — Description** *Add the following*
 31 *paragraphs to the description of the function `timer_create()`.*

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

34 If `{_POSIX_THREAD_CPUTIME}` is defined, implementations shall support E
 35 `clock_id` values representing the CPU-time clock of the calling thread. C

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

39 **14.2.2.4 Errors**

40 ⇒ **14.2.2.4 Create a Per-Process Timer — Errors** *Add the following error*
 41 *condition to the description of the function `timer_create()`:*

42 C

43 [ENOTSUP]

44 The implementation does not support the creation of a timer attached
 45 to the CPU-time clock which is specified by `clock_id` and associated
 46 with a process or thread different from the process or thread invoking
 47 `timer_create()`.

48 C

49 ⇒ **14 Clocks and Timers** *Add the following section.* C

50 **14.3 Execution Time Monitoring**

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

53 **14.3.1 CPU-time Clock Characteristics**

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

56 If `{_POSIX_THREAD_CPUTIME}` is defined, thread CPU-time clocks shall be sup- E
57 ported.

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

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

63 `CLOCK_PROCESS_CPUTIME_ID`

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

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

69 `CLOCK_THREAD_CPUTIME_ID`

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

73 **14.3.2 Accessing a Process CPU-time Clock**

74 Function: *clock_getcpuclockid()*.

75 **14.3.2.1 Synopsis**

76 `#include <sys/types.h>` C

77 `#include <time.h>`

78 `int clock_getcpuclockid (pid_t pid, clockid_t *clock_id);`

79 **14.3.2.2 Description**

80 If `{_POSIX_CPUTIME}` is defined: E

81 The *clock_getcpuclockid()* function shall return the clock ID of the CPU-time
82 clock of the process specified by *pid*. If the process described by *pid* exists
83 and the calling process has permission, the clock ID of this clock shall be
84 returned in *clock_id*.

85 If *pid* is zero, the *clock_getcpuclockid()* function shall return the clock ID of
86 the CPU-time clock of the process making the call, in *clock_id*.

87 The conditions under which one process has permission to obtain the
88 CPU-time clock ID of other processes are implementation defined.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

89 Otherwise:

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

92 **14.3.2.3 Returns**

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

95 **14.3.2.4 Errors**

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

98 [EPERM]

99 The requesting process does not have permission to access the CPU-time
100 clock for the process.

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

103 [ESRCH]

104 No process can be found corresponding to that specified by *pid*.

105 **14.3.2.5 Cross-References**

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

108 **14.3.3 Accessing a Thread CPU-time Clock**

109 Function: *pthread_getcpuclockid()*.

110 **14.3.3.1 Synopsis**

111 #include <sys/types.h> C

112 #include <time.h> C

113 #include <pthread.h>

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

115 **14.3.3.2 Description**

116 If `{_POSIX_THREAD_CPUTIME}` is defined: E

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

120 C

121 Otherwise:

122 Either the implementation shall support the *pthread_getcpuclockid()* func-
123 tion as described above or this function shall not be provided.

124 **14.3.3.3 Returns**

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

127 **14.3.3.4 Errors**

128
129 If the following condition is detected, the *pthread_getcpuclockid()* function shall
130 return the corresponding error number: C

131 [ESRCH]

132 The value specified by *thread_id* does not refer to an existing thread.

133 **14.3.3.5 Cross-References**

134 *clock_gettime()*, 14.2.1; *clock_settime()*, 14.2.1; *clock_getres()*, 14.2.1; C
135 *pthread_getcpuclockid()*, 14.3.2; *timer_create()*, 14.2.2; C

136 C

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 15: Message Passing

1 15.2 Message Passing Functions

2 15.2.4 Send a Message to a Message Queue

3 ⇒ **15.2.4 Send a Message to a Message Queue** *Add the following function to*
 4 *the list and change Function to Functions:*

5 Function: *mq_timedsend()*

6 15.2.4.1 Synopsis

7 ⇒ **15.2.4.1 Send a Message to a Message Queue — Synopsis**
 8 *Add the following #include and prototype to the end of the synopsis:* C

```
9 #include <time.h>
10 int mq_timedsend(mqd_t mqdes,
11                 const char *msg_ptr,
12                 size_t msg_len,
13                 unsigned int msg_prio,
14                 const struct timespec *abs_timeout); C
```

15 15.2.4.2 Description

16 ⇒ **15.2.4.2 Send a Message to a Message Queue — Description** *Add the fol-*
 17 *lowing text to the description:*

18 If `{_POSIX_MESSAGE_PASSING}` and `{_POSIX_TIMEOUTS}` are both defined: E

19 The *mq_timedsend()* function adds a message to the message queue
 20 specified by *mqdes* in the manner defined for the *mq_send()* function.
 21 However, if the specified message queue is full and `O_NONBLOCK` is not
 22 set in the message queue description associated with *mqdes*, the wait for
 23 sufficient room in the queue shall be terminated when the specified
 24 timeout expires. If `O_NONBLOCK` is set in the message queue descrip-
 25 tion, this function shall behave identically to *mq_send()*.

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

26 The timeout expires when the absolute time specified by *abs_timeout* C
 27 passes, as measured by the clock on which timeouts are based (that is, C
 28 when the value of that clock equals or exceeds *abs_timeout*), or if the C
 29 absolute time specified by *abs_timeout* has already been passed at the C
 30 time of the call. If the Timers option is supported, the timeout is based C
 31 on the CLOCK_REALTIME clock; if the Timers option is not supported,
 32 the timeout is based on the system clock as returned by the *time()* func-
 33 tion. The resolution of the timeout is the resolution of the clock on
 34 which it is based. The *timespec* argument is defined as a structure in
 35 the header `<time.h>`.

36 Under no circumstance shall the operation fail with a timeout if there is C
 37 sufficient room in the queue to add the message immediately. The vali-
 38 dity of the *abs_timeout* parameter need not be checked when there is
 39 sufficient room in the queue.

40 Otherwise:

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

43 15.2.4.3 Returns

44 ⇒ **15.2.4.3 Send a Message to a Message Queue — Returns** *Add the func-*
 45 *tion *mq_timedsend()* to the list of functions.*

46 15.2.4.4 Errors

47 ⇒ **15.2.4.4 Send a Message to a Message Queue — Errors** *Make the follow-*
 48 *ing changes to the discussion of error conditions:*

49 Add *mq_timedsend()* to the list of functions to which the error conditions
 50 apply.

51 Add an [ETIMEDOUT] error value with the following reason:

52 The O_NONBLOCK flag was not set when the message queue was
 53 opened, but the timeout expired before the message could be enqueued.

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

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

58
 59 Add *mq_timedsend()* to the list of functions returning [EINTR]. C

60 **15.2.4.5 Cross-References**61 ⇒ **15.2.4.5 Send a Message to a Message Queue — Cross-References**

62 Add the following cross references to the list:

63 *mq_open()*, 15.2.1; *time()* 4.5.1; <time.h>, 14.1.64 **15.2.5 Receive a Message from a Message Queue**65 ⇒ **15.2.5 Receive a Message from a Message Queue** *Add the following func-*
66 *tion to the list and change Function to Functions:*67 Function: *mq_timedreceive()*68 **15.2.5.1 Synopsis**69 ⇒ **15.2.5.1 Receive a Message from a Message Queue — Synopsis**70 *Add the following #include and prototype to the end of the synopsis:* C

71 #include <time.h>

72 int mq_timedreceive(mqd_t *mqdes*,
73 char **msg_ptr*,
74 size_t *msg_len*,
75 unsigned int **msg_prio*,
76 const struct timespec **abs_timeout*); C77 **15.2.5.2 Description**78 ⇒ **15.2.5.2 Receive a Message from a Message Queue — Description** *Add*
79 *the following text to the description:*80 If `{_POSIX_MESSAGE_PASSING}` and `{_POSIX_TIMEOUTS}` are both defined: E81 The *mq_timedreceive()* function is used to receive the oldest of the
82 highest priority messages from the message queue specified by *mqdes* as
83 in the *mq_receive()* function. However, if `O_NONBLOCK` was not
84 specified when the message queue was opened via the *mq_open()* func-
85 tion, and no message exists on the queue to satisfy the receive, the wait
86 for such a message will be terminated when the specified timeout
87 expires. If `O_NONBLOCK` is set, this function shall behave identically to
88 *mq_receive()*.89 The timeout expires when the absolute time specified by *abs_timeout* C
90 passes, as measured by the clock on which timeouts are based (that is, CCopyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

91 when the value of that clock equals or exceeds *abs_timeout*), or if the C
 92 absolute time specified by *abs_timeout* has already been passed at the C
 93 time of the call. If the Timers option is supported, the timeout is based C
 94 on the CLOCK_REALTIME clock; if the Timers option is not supported,
 95 the timeout is based on the system clock as returned by the *time()* func-
 96 tion. The resolution of the timeout is the resolution of the clock on
 97 which it is based. The *timespec* argument is defined as a structure in
 98 the header `<time.h>`.

99 Under no circumstance shall the operation fail with a timeout if a mes- C
 100 sages can be removed from the message queue immediately. The validity C
 101 of the *abs_timeout* parameter need not be checked if a message can be
 102 removed from the message queue immediately.

103 Otherwise:

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

106 15.2.5.3 Returns

107 ⇒ **15.2.5.3 Receive a Message from a Message Queue — Returns** *Add the*
 108 *mq_timedreceive()* *to the list of functions.*

109 15.2.5.4 Errors

110 ⇒ **15.2.5.4 Receive a Message from a Message Queue — Errors** *Make the*
 111 *following changes to the discussion of error conditions:*

112 Add *mq_timedreceive()* to the list of functions for both the "if occurs" error con- C
 113 ditions and the "if detected" error conditions. C

114 Add an [ETIMEDOUT] error value to the "if occurs" error conditions, with the C
 115 following reason: C

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

119 Add an [EINVAL] error value to the "if occurs" error conditions, with the follow- C
 120 ing reason: C

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

124 C

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

126 **15.2.5.5 Cross-References**

127 ⇒ **15.2.5.5 Receive a Message from a Message Queue — Cross-References**
128 *Add the following cross-references:*

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 16: Thread Management

1 **16.1 Threads**

2 **16.2.2 Thread Creation**

3 **16.2.2.2 Description**

4 ⇒ **16.2.2.2 Thread Creation — Description** *Add the following paragraph to*
5 *the description of the `pthread_create()` function:*

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

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 18: Thread Cancellation

1 18.1 Thread Cancellation Overview

2 18.1.2 Cancellation Points

3 ⇒ **18.1.2 Cancellation Points** *Add the following functions to the list of func-*
 4 *tions for which a cancellation point shall occur:*

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

6 ⇒ **18.1.2 Cancellation Points** *Add the following functions to the list of func-*
 7 *tions for which a cancellation point may also occur:*

8 *posix_fadvise(), posix_fallocate(), posix_madvise(), posix_spawn(), C*
 9 *posix_spawnnp(). C*

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Section 20: Advisory Information

1 NOTE: When this standard is approved, the section number of this chapter will be changed to
2 make it consistent with the base standard and all its approved amendments.

3 ⇒ **20 Advisory Information** *Add the following section.*

4 **20.1 I/O Advisory Information and Space Control**

5 **20.1.1 File Advisory Information**

6 Function: *posix_fadvise()*.

7 **20.1.1.1 Synopsis**

```
8 #include <sys/types.h>
9 #include <fcntl.h>
10 int posix_fadvise(int fd, off_t offset,
11                  size_t len, int advice);
```

C

12 **20.1.1.2 Description**

13 If `{_POSIX_ADVISORY_INFO}` is defined:

E

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

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

24

C

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

C

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

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

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

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

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

37 POSIX_FADV_NOREUSE specifies that the application expects to access the
38 specified data once and then not reuse it thereafter.

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

41 Otherwise:

42 Either the implementation shall support the `posix_fadvise()` function as
43 described above or this function shall not be provided.

44 **20.1.1.3 Returns**

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

47 **20.1.1.4 Errors**

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

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

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

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

53 **20.1.1.5 Cross-References** C

54 `posix_madvise()`, 20.2.1. C

55 **20.1.2 File Space Control**

56 Function: `posix_fallocate()`.

57 **20.1.2.1 Synopsis**

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```

58 #include <sys/types.h> C
59 #include <fcntl.h>
60 int posix_fallocate(int fd, off_t offset, size_t len);
61 
```

62 20.1.2.2 Description

63 If `{_POSIX_ADVISORY_INFO}` is defined: E

64 The *posix_fallocate()* function ensures that any required storage for regular C
65 file data starting at *offset* and continuing for *len* bytes is allocated on the file C
66 system storage media. If *posix_fallocate()* returns successfully, subsequent C
67 writes to the specified file data shall not fail due to the lack of free space on C
68 the file system storage media.

69 If the *offset + len* is beyond the current file size, then *posix_fallocate()* shall C
70 adjust the file size to *offset + len*. Otherwise, the file size shall not be C
71 changed. C

72 It is implementation defined whether a previous *posix_fadvise()* call C
73 influences allocation strategy.

74 Space allocated via *posix_fallocate()* shall be freed by a successful call to C
75 *creat()* or *open()* that truncates the size of the file. Space allocated via C
76 *posix_fallocate()* may be freed by a successful call to *ftruncate()* that C
77 reduces the file size to a size smaller than *offset + len*. C

78 Otherwise:

79 Either the implementation shall support the *posix_fallocate()* function as C
80 described above or this function shall not be provided. C

81 20.1.2.3 Returns

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

84 20.1.2.4 Errors

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

87	[EBADF]	The <i>fd</i> argument is not a valid file descriptor.	
88	[EBADF]	The <i>fd</i> argument references a file that was opened without write	C
89		permission.	C
90	[EFBIG]	The value of <i>offset + len</i> is greater than the maximum file size.	C
91	[EINTR]	A signal was caught during execution.	C
92	[EINVAL]	The <i>len</i> argument was zero or the <i>offset</i> argument was less than	C
93		zero.	C

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

94	[EIO]	An I/O error occurred while reading from or writing to a file system.	C
95			C
96	[ENODEV]	The <i>fd</i> argument does not refer to a regular file.	
97			C
98	[ENOSPC]	There is insufficient free space remaining on the file system storage media.	
99			
100	[ESPIPE]	The <i>fd</i> argument is associated with a pipe or FIFO.	C

101 **20.1.2.5 Cross-References**

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

103 **20.2 Memory Advisory Information and Alignment Control**

104 **20.2.1 Memory Advisory Information**

105 Function: *posix_madvise()*.

106 **20.2.1.1 Synopsis**

```
107 #include <sys/types.h> C
108 #include <sys/mman.h>
109 int posix_madvise(void *addr, size_t len, int advice);
```

110 **20.2.1.2 Description**

111 If `{_POSIX_ADVISORY_INFO}` is defined and either `{_POSIX_MAPPED_FILES}` is E
 112 defined or `{_POSIX_SHARED_MEMORY_OBJECTS}` is defined: E

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

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

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

125 C

193	20.2.2.5 Cross-References	C
194	<i>free()</i> , 8.1; <i>malloc()</i> , 8.1.	C

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Annex A (informative) Bibliography

1	A.2 Other Standards	C
2	⇒ A.2 Other Standards <i>Add the following to the end of subclause A.2, with an</i>	C
3	<i>appropriate reference number:</i>	C
4	{B1} ISO/IEC 14519:1998, <i>POSIX Ada Language Interfaces—Binding for Sys-</i>	C
5	<i>tem Application Interfaces (API) including Realtime Extensions.</i>	C
6	A.3 Historical Documentation and Introductory Texts	C
7	⇒ A.3 Historical Documentation and Introductory Texts <i>Add the following</i>	C
8	<i>to the end of subclause A.3, with an appropriate reference number:</i>	C
9	{B2} Sprunt, B.; Sha, L.; and Lehoczky, J.P. "Aperiodic Task Scheduling for	C
10	Hard Real-Time Systems". <i>The Journal of Real-Time Systems, Vol. 1,</i>	C
11	<i>1989, pages 27-60.</i>	C

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Annex B (informative)

Rationale and Notes

1

2 **B.2 Definitions and General Requirements**

3 **B.2.3 General Concepts**

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

6 **B.2.3.1 execution time measurement**

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

14 **B.3 Process Primitives**

15 **B.3.1 Process Creation and Execution**

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

17 **B.3.1.4 Spawn File Actions**

18 A spawn file actions object may be initialized to contain an ordered sequence of D
 19 close, dup2, and open operations to be used by *posix_spawn()* or *posix_spawnp()* D
 20 arrive at the set of open file descriptors inherited by the spawned process from the D
 21 set of open file descriptors in the parent at the time of the *posix_spawn()* or D
 22 *posix_spawnp()* call. It had been suggested that the close and dup2 operations D

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

23 alone are sufficient to rearrange file descriptors, and that files which need be D
 24 opened for use by the spawned process can be handled either by having the calling D
 25 process open them before the *posix_spawn()* or *posix_spawnnp()* call (and close D
 26 them after), or by passing file names to the spawned process (in *argv*) so that it D
 27 may open them itself. The working group recommends that applications use one of D
 28 these two methods when practical, since detailed error status on a failed open D
 29 operation is always available to the application this way. However, the working D
 30 group feels that allowing a spawn file actions object to specify open operations is D
 31 still appropriate because: D

- 32 (1) It is consistent with equivalent POSIX.5 functionality (see the discussion D
 33 on compatibility with POSIX.5, in B.3.1.6). D
- 34 (2) It supports the I/O redirection paradigm commonly employed by POSIX D
 35 programs designed to be invoked from a shell. When such a program is D
 36 the child process, it may not be designed to open files on its own. D
- 37 (3) It allows file opens that might otherwise fail or violate file D
 38 ownership/access rights if executed by the parent process. D

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

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

48 Additionally, if a parent is executed from a file having a “set-user-id” mode bit set D
 49 and the POSIX_SPAWN_RESETEUIDS flag is set in the spawn attributes, a file created D
 50 within the parent process will (possibly incorrectly) have the parent’s effective D
 51 user id as its owner whereas a file created via an open action during D
 52 *posix_spawn()* or *posix_spawnnp()* will have the parent’s real id as its owner; and D
 53 an open by the parent process may successfully open a file to which the real user D
 54 should not have access or fail to open a file to which the real user should have D
 55 access. D

56 ***File Descriptor Mapping Rationale***

57 C
 58 The working group had originally proposed using an array which specified the C
 59 mapping of child file descriptors back to those of the parent. It was pointed out by C
 60 the ballot group that it is not possible to re-shuffle file descriptors arbitrarily in a C
 61 library implementation of *posix_spawn()* or *posix_spawnnp()* without provision for C
 62 one or more spare file descriptor entries (which simply may not be available). Such C
 63 an array requires that an implementation develop a complex strategy to achieve C
 64 the desired mapping without inadvertently closing the wrong file descriptor at the C
 65 wrong time. C

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

74 **B.3.1.5 Spawn Attributes** D

75 The original spawn interface proposed in this standard, defined the attributes that D
 76 specify the inheritance of process attributes across a spawn operation as a struc- D
 77 ture. In order to be able to separate optional individual attributes under their D
 78 appropriate options (i.e., the `spawn-schedparam` and `spawn-schedpolicy` D
 79 attributes depending upon the Process scheduling option), and also for extensibil- D
 80 ity and consistency with the newer posix interfaces, the attributes interface has D
 81 been changed to an opaque datatype. This interface now consists of the type D
 82 `posix_spawnattr_t`, representing a spawn attributes object, together with associ- D
 83 ated functions to initialize or destroy the attributes object, and to set or get each D
 84 individual attribute. Although the new object-oriented interface is more verbose D
 85 than the original structure, it is simple to use, more extensible, and easy to imple- D
 86 ment. D

87 **B.3.1.6 Spawn a Process**

88 The POSIX `fork()` function is difficult or impossible to implement without swapping
 89 or dynamic address translation. Since:

- 90 — Swapping is generally too slow for a realtime environment,
- 91 — dynamic address translation is not available everywhere POSIX might be
 92 useful,
- 93 — and processes are too useful to simply option out of POSIX whenever it must
 94 run without address translation or other MMU services,

95 POSIX needs process creation and file execution primitives that can be efficiently
 96 implemented without address translation or other MMU services.

97 We shall call this function `posix_spawn()`. A closely related function, C
 98 `posix_spawnnp()`, is included for completeness. C

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

104 This view of the role of `posix_spawn()` and `posix_spawnnp()` influenced the design of C
 105 their API. It does not attempt to provide the full functionality of `fork()` / `exec` in C
 106 which arbitrary user specified operations of any sort are permitted between the C

107 creation of the child process and the execution of the new process image; any C
 108 attempt to reach that level would need to provide a programming language as C
 109 parameters. Instead, *posix_spawn()* and *posix_spawnp()* are process creation C
 110 primitives like the `Start_Process` and `Start_Process_Search` Ada language C
 111 bindings in ISO/IEC 14519:1998 {B1} package `POSIX_Process_Primitives` and also C
 112 like those in many operating systems that are not UNIX¹⁾ systems, but with some D
 113 POSIX specific additions. D

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

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

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

127 The intention is that the normal file descriptor inheritance across *fork()*, the sub- C
 128 sequent effect of the specified spawn file actions, and the normal file descriptor C
 129 inheritance across one of the *exec* family of functions should fully specify open file C
 130 inheritance. The implementation need make no decisions regarding the set of C
 131 open file descriptors when the child process image begins execution, those deci- C
 132 sions having already been made by the caller and expressed as the set of open file C
 133 descriptors and their `FD_CLOEXEC` flags at the time of the call and the spawn file C
 134 actions object specified in the call. We have been assured that in cases where the C
 135 POSIX `Start_Process` Ada primitives have been implemented in a library, this C
 136 method of controlling file descriptor inheritance may be implemented very easily. C
 137 See Figure B-1 for a crude, but workable, C language implementation. C

138 We can identify several problems with *posix_spawn()* and *posix_spawnp()* but C
 139 there does not appear to be a solution that introduces fewer problems. C

140 Environment modification for child process attributes not specifiable via the *attrp* C
 141 or *file_actions* arguments must be done in the parent process, and since the C
 142 parent generally wants to save its context, it is more costly than similar func- C
 143 tionality with *fork()* / *exec*. It is also complicated to modify the environment of a C
 144 multi-threaded process temporarily, since all threads must agree when it is safe C
 145 for the environment to be changed. However, this cost is only borne by those invo- C
 146 cations of *posix_spawn()* and *posix_spawnp()* that use the additional functionality. C
 147 Since extensive modifications are not the usual case, and are particularly unlikely C

148 _____
 149 1) UNIX is a registered trademark of The Open Group in the US and other countries. D

150 in time-critical code, keeping much of the environment control out of C
 151 *posix_spawn()* and *posix_spawnp()* is appropriate design. C

152 The *posix_spawn()* and *posix_spawnp()* functions do not have all the power of C
 153 *fork()* / *exec*. This is to be expected. The *fork()* function is a wonderfully powerful C
 154 operation. We do not expect to duplicate its functionality in a simple, fast function
 155 with no special hardware requirements. It is worth noting that *posix_spawn()* C
 156 and *posix_spawnp()* are very similar to the process creation operations on many D
 157 operating systems that are not UNIX systems. D
 158 C

159 **Requirements**

160 The requirements for *posix_spawn()* and *posix_spawnp()* are: C

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

163 Additional goals are:

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

170 **Two-Syntax Rationale**

171 POSIX *exec* has several calling sequences with approximately the same functional-
 172 ity. These appear to be required for compatibility with existing practice. Since
 173 the existing practice for the *posix_spawn* functions is otherwise substantially
 174 unlike POSIX, we feel that simplicity outweighs compatibility. There are, there-
 175 fore, only two names for the *posix_spawn* functions.

176 The parameter list does not differ between *posix_spawn()* and *posix_spawnp()*;
 177 *posix_spawnp()* interprets the second parameter more elaborately than D
 178 *posix_spawn()*. D
 179 C

180 **Compatibility with POSIX.5** `POSIX_Process_Primitives.Start_Process`

181 The `Start_Process` and `Start_Process_Search` procedures from ISO/IEC
 182 14519:1998 {B1}, the Ada Language Binding to POSIX.1, encapsulate *fork()* and
 183 *exec* functionality in a manner similar to that of *posix_spawn()* and C
 184 *posix_spawnp()*. Originally, in keeping with our simplicity goal, the working C
 185 group had limited the capabilities of *posix_spawn()* and *posix_spawnp()* to a sub- C
 186 set of the capabilities of `Start_Process` and `Start_Process_Search`; certain C
 187 non-default capabilities were not supported. However, based on suggestions by the C
 188 ballot group to improve file descriptor mapping or drop it, and on the advice of an C
 189 Ada Bindings working group member, the working group decided that C
 190 *posix_spawn()* and *posix_spawnp()* should be sufficiently powerful to implement C

191 `Start_Process` and `Start_Process_Search`. The rationale is that if the Ada C
 192 language binding to such a primitive had already been approved as an IEEE stan- C
 193 dard, there can be little justification for not approving the functionally equivalent C
 194 parts of a C binding. The only three capabilities provided by `posix_spawn()` and C
 195 `posix_spawnp()` that are not provided by `Start_Process` and `Start_Process_-` C
 196 `Search` are optionally specifying the child's process group id, the set of signals to C
 197 be reset to default signal handling in the child process, and the child's scheduling C
 198 policy and parameters. C

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

206 **Process Group**

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

211 **Threads**

212 Without the `posix_spawn()` and `posix_spawnp()` functions, systems without C
 213 address translation can still use threads to give an abstraction of concurrency. In
 214 many cases, thread creation suffices, but it is not always a good substitute. The
 215 `posix_spawn()` and `posix_spawnp()` functions are considerably “heavier” than C
 216 thread creation. Processes have several important attributes that threads do not.
 217 Even without address translation, a process may have base-and-bound memory
 218 protection. Each process has a process environment including security attributes
 219 and file capabilities, and powerful scheduling attributes specified by POSIX.1 and
 220 POSIX.1b. Processes abstract the behavior of non-uniform-memory-architecture
 221 multi-processors better than threads, and they are more convenient to use for
 222 activities that are not closely linked.

223 The `posix_spawn()` and `posix_spawnp()` functions may not bring support for multi- C
 224 ple processes to every configuration. Process creation is not the only piece of
 225 operating system support required to support multiple processes. The total cost of
 226 support for multiple processes may be quite high in some circumstances. Existing
 227 practice shows that support for multiple processes is uncommon and threads are
 228 common among “tiny kernels.” There should, therefore, probably continue to be
 229 AEPs for operating systems with only one process.

230 **Asynchronous Error Notification Rationale**

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

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

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

242 The working group had proposed using two additional macros to interpret
 243 *stat_val*: The first, WIFSPAWNFAIL, would have detected a status that indicated
 244 that the child exited because of an error detected during the *posix_spawn()* or
 245 *posix_spawnnp()* operations rather than during actual execution of the child pro-
 246 cess image; the second, WSPAWNERRNO, would have extracted the error value if
 247 WIFSPAWNFAIL indicated a failure. Unfortunately, the ballot group strongly
 248 opposed this because it would make a library implementation of *posix_spawn()* or
 249 *posix_spawnnp()* dependent on kernel modifications to *waitpid()* to be able to
 250 embed special information in *stat_val* to indicate a spawn failure.

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

264 Thus, no special macros are available to isolate asynchronous *posix_spawn()* or
 265 *posix_spawnnp()* errors. Instead, errors detected by the *posix_spawn()* or
 266 *posix_spawnnp()* operations in the context of the child process before the new pro-
 267 cess image executes are reported by setting the child's exit status to 127. The cal-
 268 ling process may use the WIFEXITED and WEXITSTATUS macros on the *stat_val*
 269 stored by the *wait()* or *waitpid()* functions to detect spawn failures to the extent
 270 that other status values with which the child process image may exit (before the
 271 parent can conclusively determine that the child process image has begun execu-
 272 tion) are distinct from exit status 127.

273 **Library Implementation of Spawn**

274 The *posix_spawn()* or *posix_spawnnp()* operation is enough to:

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

280 — Run the child at any priority in the domain of the parent.

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

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

```

290
291 #include <sys/types.h> C
292 #include <stdlib.h> C
293 #include <stdio.h> C
294 #include <unistd.h> C
295 #include <sched.h> C
296 #include <fcntl.h> C
297 #include <signal.h> C
298 #include <errno.h> C
299 #include <string.h> C
300 #include <signal.h> D

301 /*#include <spawn.h>*/ C
302 /***** C
303 /*Things that could be defined in spawn.h*/ C
304 /***** C
305 typedef struct D
306 { C
307     short posix_attr_flags; C
308     #define POSIX_SPAWN_SETPGROUP 0x1 C
309     #define POSIX_SPAWN_SETSIGMASK 0x2 C
310     #define POSIX_SPAWN_SETSIGDEF 0x4 C
311     #define POSIX_SPAWN_SETSCHEDULER 0x8 C
312     #define POSIX_SPAWN_SETSCHEDPARAM 0x10 C
313     #define POSIX_SPAWN_RESETPIDS 0x20 C
314     pid_t posix_attr_pgroup; C
315     sigset_t posix_attr_sigmask; C
316     sigset_t posix_attr_sigdefault; C
317     int posix_attr_schedpolicy; C
318     struct sched_param posix_attr_schedparam; C
319     } posix_spawnattr_t; D

320 typedef char *posix_spawn_file_actions_t; C

321 int posix_spawn_file_actions_init( C
322     posix_spawn_file_actions_t *file_actions); C
323 int posix_spawn_file_actions_destroy( C
324     posix_spawn_file_actions_t *file_actions); C
325 int posix_spawn_file_actions_addclose( C
326     posix_spawn_file_actions_t *file_actions, C
327     int fildes); C

```

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

```

328 int posix_spawn_file_actions_adddup2(          C
329     posix_spawn_file_actions_t *file_actions,  C
330     int fildes, int newfildes);               C
331 int posix_spawn_file_actions_addopen(         C
332     posix_spawn_file_actions_t *file_actions,  C
333     int fildes, const char *path, int oflag,   C
334     mode_t mode);                             C
335 int posix_spawnattr_init (                   D
336     posix_spawnattr_t *attr);                 D
337 int posix_spawnattr_destroy (                D
338     posix_spawnattr_t *attr);                 D
339 int posix_spawnattr_getflags (               D
340     const posix_spawnattr_t *attr,            D
341     short *flags);                            D
342 int posix_spawnattr_setflags (               D
343     posix_spawnattr_t *attr,                  D
344     short flags);                             D
345 int posix_spawnattr_getpgroup (              D
346     const posix_spawnattr_t *attr,            D
347     pid_t *pgroup);                           D
348 int posix_spawnattr_setpgroup (              D
349     posix_spawnattr_t *attr,                  D
350     pid_t pgroup);                            D
351 int posix_spawnattr_getschedpolicy (         D
352     const posix_spawnattr_t *attr,            D
353     int *schedpolicy);                         D
354 int posix_spawnattr_setschedpolicy (         D
355     posix_spawnattr_t *attr,                  D
356     int schedpolicy);                         D
357 int posix_spawnattr_getschedparam (          D
358     const posix_spawnattr_t *attr,            D
359     struct sched_param *schedparam);           D
360 int posix_spawnattr_setschedparam (          D
361     posix_spawnattr_t *attr,                  D
362     const struct sched_param *schedparam);     D
363 int posix_spawnattr_getsigmask (             D
364     const posix_spawnattr_t *attr,            D
365     sigset_t *sigmask);                       D
366 int posix_spawnattr_setsigmask (             D
367     posix_spawnattr_t *attr,                  D
368     const sigset_t *sigmask);                 D
369 int posix_spawnattr_getdefault (             D
370     const posix_spawnattr_t *attr,            D
371     sigset_t *sigdefault);                    D
372 int posix_spawnattr_setdefault (             D
373     posix_spawnattr_t *attr,                  D
374     const sigset_t *sigdefault);              D
375 int posix_spawn(                             C
376     pid_t *pid,                               C
377     const char *path,                          C
378     const posix_spawn_file_actions_t *file_actions,  C
379     const posix_spawnattr_t *attrp,           D
380     char * const argv[],                       C
381     char * const envp[]);                     C

```

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```

382 int posix_spawn(
383     pid_t *pid,
384     const char *file,
385     const posix_spawn_file_actions_t *file_actions,
386     const posix_spawnattr_t *attrp,
387     char * const argv[],
388     char * const envp[]);

389 /*****/
390 /*Example posix_spawn() library routine*/
391 /*****/
392 int posix_spawn(pid_t *pid,
393     const char *path,
394     const posix_spawn_file_actions_t *file_actions,
395     const posix_spawnattr_t *attrp,
396     char * const argv[],
397     char * const envp[])
398 {
399     /*Create process*/
400     if((*pid=fork()) == (pid_t)0)
401     {
402         /*This is the child process*/
403         /*Worry about process group*/
404         if(attrp->posix_attr_flags & POSIX_SPAWN_SETPGROUP)
405         {
406             /*Override inherited process group*/
407             if(setpgid(0, attrp->posix_attr_pgroup) != 0)
408             {
409                 /*Failed*/
410                 exit(127);
411             }
412         }

413         /*Worry about process signal mask*/
414         if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGMASK)
415         {
416             /*Set the signal mask (can't fail)*/
417             sigprocmask(SIG_SETMASK, &attrp->posix_attr_sigmask,
418                 NULL);
419         }

420         /*Worry about resetting effective user and group IDs*/
421         if(attrp->posix_attr_flags & POSIX_SPAWN_RESETPIDS)
422         {
423             /*None of these can fail for this case.*/
424             setuid(getuid());
425             setgid(getgid());
426         }

427         /*Worry about defaulted signals*/
428         if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGDEF)
429         {
430             struct sigaction deflt;
431             sigset_t all_signals;

```

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.


```

432         int s;                                     C
433
434         /*Construct default signal action*/         C
435         deflt.sa_handler = SIG_DFL;                C
436         deflt.sa_flags = 0;                         C
437
438         /*Construct the set of all signals*/         C
439         sigfillset(&all_signals);                  C
440
441         /*Loop for all signals*/                   C
442         for(s=0; sigismember(&all_signals,s); s++) C
443         {                                           C
444             /*Signal to be defaulted?*/           C
445             if(sigismember(&attrp->posix_attr_sigdefault,s)) C
446             {                                       C
447                 /*Yes - default this signal*/     C
448                 if(sigaction(s, &deflt, NULL) == -1) C
449                 {                                       C
450                     /*Failed*/                     C
451                     exit(127);                       D
452                 }                                       C
453             }                                       C
454
455         }                                           C
456
457         /*Worry about the fds if we are to map them*/ C
458         if(file_actions != NULL)                   C
459         {                                           C
460             /*Loop for all actions in object *file_actions*/ C
461             /*(implementation dives beneath abstraction)*/ C
462             char *p = *file_actions;                C
463             while(*p != '\0')                       C
464             {                                       C
465                 if(strncmp(p,"close(",6) == 0)     C
466                 {                                       C
467                     int fd;                          C
468                     if(sscanf(p+6,"%d",&fd) != 1) C
469                     {                                       C
470                         exit(127);                       D
471                     }                                       C
472                     if(close(fd) == -1) exit(127);     D
473                 }                                       C
474                 else if(strncmp(p,"dup2(",5) == 0) C
475                 {                                       C
476                     int fd,newfd;                     C
477                     if(sscanf(p+5,"%d,%d",&fd,&newfd) != 2) C
478                     {                                       C
479                         exit(127);                       D
480                     }                                       C
481                     if(dup2(fd, newfd) == -1) exit(127); D
482                 }                                       C
483                 else if(strncmp(p,"open(",5) == 0) C
484                 {                                       C
485                     int fd,oflag;                     C

```

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```

483         mode_t mode;
484         int tempfd;
485         char path[1000]; /*should be dynamic*/
486         char *q;
487         if(sscanf(p+5,"%d",&fd) != 1)
488             {
489                 exit(127);
490             }
491         p = strchr(p, ',') + 1;
492         q = strchr(p, '*');
493         if(q == NULL) exit(127);
494         strncpy(path, p, q-p);
495         path[q-p] = '\0';
496         if(sscanf(q+1,"%o,%o",&oflag,&mode)!=2)
497             {
498                 exit(127);
499             }
500         if(close(fd) == -1)
501             {
502                 if(errno != EBADF) exit(127);
503             }
504         tempfd = open(path, oflag, mode);
505         if(tempfd == -1) exit(127);
506         if(tempfd != fd)
507             {
508                 if(dup2(tempfd,fd) == -1)
509                     {
510                         exit(127);
511                     }
512                 if(close(tempfd) == -1)
513                     {
514                         exit(127);
515                     }
516             }
517         }
518     else
519     {
520         exit(127);
521     }
522     p = strchr(p, ',') + 1;
523 }
524 }

525     /*Worry about setting new scheduling policy and parameters*/
526     if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDULER)
527     {
528         if(sched_setscheduler(0, attrp->posix_attr_schedpolicy,
529             &attrp->posix_attr_schedparam) == -1)
530             {
531                 exit(127);
532             }
533     }

534     /*Worry about setting only new scheduling parameters*/

```

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```

535         if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDPARAM)           C
536             {                                                             C
537                 if(sched_setparam(0, &attrp->posix_attr_schedparam)==-1)   C
538                     {                                                     C
539                         exit(127);                                         D
540                     }                                                     C
541             }                                                             C

542         /*Now execute the program at path*/                               C
543         /*Any fd that still has FD_CLOEXEC set will be closed*/           C
544         execve(path, argv, envp);                                         C
545         exit(127); /*exec failed*/                                         D
546     }                                                                     C
547     else                                                                    C
548     {                                                                       C
549         /*This is the parent (calling) process*/                          C
550         if((int)pid == -1) return errno;                                    C
551         return 0;                                                           C
552     }                                                                       C
553 }                                                                           C

554 /******                                                                    C
555 /* Here is a crude but effective implementation of the */                 C
556 /* file action object operators which store actions as */                 C
557 /* concatenated token separated strings.                                 */ C
558 /******                                                                    C
559 /*Create object with no actions.*/                                       C
560 int posix_spawn_file_actions_init(                                       C
561     posix_spawn_file_actions_t *file_actions)                             C
562     {                                                                       C
563         *file_actions = malloc(sizeof(char));                               C
564         if(*file_actions == NULL) return ENOMEM;                           C
565         strcpy(*file_actions, "");                                         C
566         return 0;                                                           C
567     }                                                                       C

568 /*Free object storage and make invalid.*/                                  C
569 int posix_spawn_file_actions_destroy(                                     C
570     posix_spawn_file_actions_t *file_actions)                             C
571     {                                                                       C
572         free(*file_actions);                                               C
573         *file_actions = NULL;                                              C
574         return 0;                                                           E
575     }                                                                       C

576 /*Add a new action string to object.*/                                     C
577 static int add_to_file_actions(                                           C
578     posix_spawn_file_actions_t *file_actions,                             C
579     char *new_action)                                                      C
580     {                                                                       C
581         *file_actions = realloc                                           C
582             (*file_actions, strlen(*file_actions)+strlen(new_action)+1);   C
583         if(*file_actions == NULL) return ENOMEM;                           C
584         strcat(*file_actions, new_action);                                  C

```

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```

585     return 0;
586     }

587 /*Add a close action to object.*/
588 int posix_spawn_file_actions_addclose(
589     posix_spawn_file_actions_t *file_actions,
590     int fildes)
591     {
592     char temp[100];
593     sprintf(temp, "close(%d)", fildes);
594     return add_to_file_actions(file_actions, temp);
595     }

596 /*Add a dup2 action to object.*/
597 int posix_spawn_file_actions_adddup2(
598     posix_spawn_file_actions_t *file_actions,
599     int fildes, int newfildes)
600     {
601     char temp[100];
602     sprintf(temp, "dup2(%d,%d)", fildes, newfildes);
603     return add_to_file_actions(file_actions, temp);
604     }
605
606 /*Add an open action to object.*/
607 int posix_spawn_file_actions_addopen(
608     posix_spawn_file_actions_t *file_actions,
609     int fildes, const char *path, int oflag,
610     mode_t mode)
611     {
612     char temp[100];
613     sprintf(temp, "open(%d,%s*%o,%o)", fildes, path, oflag, mode);
614     return add_to_file_actions(file_actions, temp);
615     }

616 /******
617 /* Here is a crude but effective implementation of the */
618 /* spawn attributes object functions which manipulate */
619 /* the individual attributes. */
620 /******
621 /*Initialize object with default values.*/
622 int posix_spawnattr_init (
623     posix_spawnattr_t *attr)
624     {
625     attr->posix_attr_flags=0;
626     attr->posix_attr_pgroup=0;
627     /* Default value of signal mask is the parent's signal mask */
628     /* other values are also allowed */
629     sigprocmask(0,NULL,&attr->posix_attr_sigmask);
630     sigemptyset(&attr->posix_attr_sigdefault);
631     /* Default values of scheduling attr. inherited from the parent */
632     /* other values are also allowed */
633     attr->posix_attr_schedpolicy=sched_getscheduler(0);
634     sched_getparam(0,&attr->posix_attr_schedparam);
635     return 0;

```

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

```

636     }
637 int posix_spawnattr_destroy (
638     posix_spawnattr_t *attr)
639     {
640     /* No action needed */
641     return 0;
642     }

643 int posix_spawnattr_getflags (
644     const posix_spawnattr_t *attr,
645     short *flags)
646     {
647     *flags=attr->posix_attr_flags;
648     return 0;
649     }

650 int posix_spawnattr_setflags (
651     posix_spawnattr_t *attr,
652     short flags)
653     {
654     attr->posix_attr_flags=flags;
655     return 0;
656     }

657 int posix_spawnattr_getpgroup (
658     const posix_spawnattr_t *attr,
659     pid_t *pgroup)
660     {
661     *pgroup=attr->posix_attr_pgroup;
662     return 0;
663     }

664 int posix_spawnattr_setpgroup (
665     posix_spawnattr_t *attr,
666     pid_t pgroup)
667     {
668     attr->posix_attr_pgroup=pgroup;
669     return 0;
670     }

671 int posix_spawnattr_getschedpolicy (
672     const posix_spawnattr_t *attr,
673     int *schedpolicy)
674     {
675     *schedpolicy=attr->posix_attr_schedpolicy;
676     return 0;
677     }

678 int posix_spawnattr_setschedpolicy (
679     posix_spawnattr_t *attr,
680     int schedpolicy)
681     {
682     attr->posix_attr_schedpolicy=schedpolicy;

```

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```

683     return 0;
684     }

685 int posix_spawnattr_getschedparam (
686     const posix_spawnattr_t *attr,
687     struct sched_param *schedparam)
688     {
689     *schedparam=attr->posix_attr_schedparam;
690     return 0;
691     }

692 int posix_spawnattr_setschedparam (
693     posix_spawnattr_t *attr,
694     const struct sched_param *schedparam)
695     {
696     attr->posix_attr_schedparam=*schedparam;
697     return 0;
698     }

699 int posix_spawnattr_getsigmask (
700     const posix_spawnattr_t *attr,
701     sigset_t *sigmask)
702     {
703     *sigmask=attr->posix_attr_sigmask;
704     return 0;
705     }

706 int posix_spawnattr_setsigmask (
707     posix_spawnattr_t *attr,
708     const sigset_t *sigmask)
709     {
710     attr->posix_attr_sigmask=*sigmask;
711     return 0;
712     }

713 int posix_spawnattr_getsigdefault (
714     const posix_spawnattr_t *attr,
715     sigset_t *sigdefault)
716     {
717     *sigdefault=attr->posix_attr_sigdefault;
718     return 0;
719     }

720 int posix_spawnattr_setsigdefault (
721     posix_spawnattr_t *attr,
722     const sigset_t *sigdefault)
723     {
724     attr->posix_attr_sigdefault=*sigdefault;
725     return 0;
726     }
727

```

728

Figure B-1 – *posix_spawn()* Equivalent

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

```

732 _____ C
733 /* To redirect new standard output (fd 1) to a file, */ C
734 /* and redirect new standard input (fd 0) from my fd socket_pair[1], */ C
735 /* and close my fd socket_pair[0] in the new process. */ C
736 posix_spawn_file_actions_t file_actions; C
737 posix_spawn_file_actions_init (&file_actions); C
738 posix_spawn_file_actions_addopen (&file_actions, 1, "newout", ...); C
739 posix_spawn_file_actions_dup2 (&file_actions, socket_pair[1], 0); C
740 posix_spawn_file_actions_close (&file_actions, socket_pair[0]); C
741 posix_spawn_file_actions_close (&file_actions, socket_pair[1]); C
742 posix_spawn(..., &file_actions, ...) C
743 posix_spawn_file_actions_destroy (&file_actions); C
744 _____ C
  
```

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

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

```

747 _____
748 Save = getuid();
749 setuid(newid);
750 posix_spawn(...)
751 setuid(Save);
752 _____
  
```

753 **Figure B-3 – Spawning a new Userid Process**

754 **B.13 Execution Scheduling**

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

756 **B.13.3 Sporadic Server Scheduling Policy**

757 The sporadic server is a mechanism defined for scheduling aperiodic activities in
 758 time-critical realtime systems. This mechanism reserves a certain bounded
 759 amount of execution capacity for processing aperiodic events at a high priority
 760 level. Any aperiodic events that cannot be processed within the bounded amount
 761 of execution capacity are executed in the background at a low priority level. Thus,
 762 a certain amount of execution capacity can be guaranteed to be available for pro-
 763 cessing periodic tasks, even under burst conditions in the arrival of aperiodic pro-
 764 cessing requests (i.e. a large number of requests in a short time interval). The
 765 sporadic server also simplifies the schedulability analysis of the realtime system,

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

766 because it allows aperiodic processes or threads to be treated as if they were
767 periodic. The sporadic server was first described by Sprunt, et al. {B2}.

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

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

793 **B.13.3.1 Scheduling Aperiodic Activities (rationale)**

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

- 798 — The effects of an aperiodic activity on the response time of lower priority
799 activities must be controllable and predictable.
- 800 — The system must provide the fastest possible response time to aperiodic
801 events.
- 802 — It must be possible to take advantage of all the available processing
803 bandwidth not needed by time-critical activities to enhance average-case
804 response times to aperiodic events.

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

- 807 — Background processing consists of assigning a very low priority to the pro-
808 cessing of aperiodic events. It utilizes all the available bandwidth in the

809 system that has not been consumed by higher priority threads. However, it
810 is very difficult, or impossible, to meet requirements on average-case
811 response time, because the aperiodic entity has to wait for the execution of
812 all other entities which have higher priority.

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

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

837 The sporadic server scheduling algorithm combines the predictability of the pol-
838 ling approach with the short response times of the direct event execution. Thus, it
839 allows systems to meet an important class of application requirements that cannot
840 be met by using the traditional approaches. Multiple sporadic servers with
841 different attributes can be applied to the scheduling of multiple classes of
842 aperiodic events, each with different kinds of timing requirements, such as indivi-
843 dual deadlines, average response times, etc. It also has many other interesting
844 applications for realtime, such as scheduling producer/consumer tasks in
845 time-critical systems, limiting the effects of faults on the estimation of task
846 execution-time requirements, etc.

847 **B.13.3.2 Existing Practice**

848 The sporadic server has been used in different kinds of applications, including mil-
849 itary avionics, robot control systems, industrial automation systems, etc. There
850 are examples of many systems that cannot be successfully scheduled using the
851 classic approaches such as direct event execution, or polling, and are schedulable
852 using a sporadic server scheduler. The sporadic server algorithm itself can suc-
853 cessfully schedule all systems scheduled with direct event execution or polling.

854 The sporadic server scheduling policy has been implemented as a commercial pro-
855 duct in the run-time system of the Verdex Ada compiler. There are also many
856 applications that have used a much less efficient application-level sporadic server.
857 These real-time applications would benefit from a sporadic server scheduler imple-
858 mented at the scheduler level.

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

860 The sporadic server interface described in this section requires the sporadic server
861 policy to be implemented at the same level as the scheduler. This means that the
862 process sporadic server shall be implemented at the kernel level and the thread
863 sporadic server policy shall be implemented at the same level as the thread
864 scheduler, i.e. kernel or library level.

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

876 **B.13.3.4 Range of Scheduling Priorities**

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

889 **B.13.3.5 Dynamically Setting the Sporadic Server Policy**

890 Several members of the Working Group requested that implementations should
891 not be required to support dynamically setting the sporadic server scheduling pol-
892 icy for a thread. The reason is that this policy may have a high overhead for
893 library-level implementations of threads, and if threads are allowed to dynami-
894 cally set this policy this overhead can be experienced even if the thread does not
895 use that policy. By disallowing the dynamic setting of the sporadic server

896 scheduling policy, these implementations can accomplish efficient scheduling for
897 threads using other policies. If a strictly conforming application needs to use the
898 sporadic server policy, and is therefore willing to pay the overhead, it must set
899 this policy at the time of thread creation.

900 **B.13.3.6 Limitation of the Number of Pending Replenishments**

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

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

921 **B.14 Clocks and Timers**

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

923 **B.14.3 Execution Time Monitoring**

924 **B.14.3.1 Introduction**

925 The main goals of the execution time monitoring facilities defined in this chapter
926 are to measure the execution time of processes and threads and to allow an appli-
927 cation to establish CPU time limits for these entities. The analysis phase of
928 time-critical realtime systems often relies on the measurement of execution times
929 of individual threads or processes to determine whether the timing requirements
930 will be met. Also, performance analysis techniques for soft deadline realtime sys-
931 tems rely heavily on the determination of these execution times. The execution
932 time monitoring functions provide application developers with the ability to

933 measure these execution times on-line and open the possibility of dynamic
934 execution-time analysis and system reconfiguration, if required. The second goal
935 of allowing an application to establish execution time limits for individual
936 processes or threads and detecting when they overrun allows program robustness
937 to be increased by enabling on-line checking of the execution times. If errors are
938 detected — possibly because of erroneous program constructs, the existence of
939 errors in the analysis phase, or a burst of event arrivals — on-line detection and
940 recovery is possible in a portable way. This feature can be extremely important for
941 many time-critical applications. Other applications require trapping CPU-time
942 errors as a normal way to exit an algorithm; for instance, some realtime artificial
943 intelligence applications trigger a number of independent inference processes of
944 varying accuracy and speed, limit how long they can run, and pick the best answer
945 available when time runs out. In many periodic systems, overrun processes are
946 simply restarted in the next resource period, after necessary end-of-period actions
947 have been taken. This allows algorithms that are inherently data-dependent to be
948 made predictable.

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

956 **B.14.3.2 Execution Time Monitoring Interface**

957 The clock and timer interface defined in POSIX.1b (Section 14) only defines one
958 clock, which measures wall-clock time. The requirements for measuring execution
959 time of processes and threads, and setting limits to their execution time by detect-
960 ing when they overrun, can be accomplished with that interface if a new kind of
961 clock is defined. These new clocks measure execution time, and one is associated
962 with each process and with each thread. The clock functions currently defined in
963 POSIX.1b can be used to read and set these CPU-time clocks, and timers can be
964 created using these clocks as their timing base. These timers can then be used to
965 send a signal when some specified execution time has been exceeded. The
966 CPU-time clocks of each process or thread can be accessed by using the symbols
967 `CLOCK_PROCESS_CPUTIME_ID`, or `CLOCK_THREAD_CPUTIME_ID`.

968 The clock and timer interface defined in POSIX.1b and extended with the new kind
969 of CPU-time clock would only allow processes or threads to access their own
970 CPU-time clocks. However, many realtime systems require the possibility of moni-
971 toring the execution time of processes or threads from independent monitoring
972 entities. In order to allow applications to construct independent monitoring enti-
973 ties that do not require cooperation from or modification of the monitored entities,
974 two functions have been defined in this chapter: `clock_getcpuclockid()`, for access-
975 ing CPU-time clocks of other processes, and `pthread_getcpuclockid()`, for accessing
976 CPU-time clocks of other threads. These functions return the clock identifier asso-
977 ciated with the process or thread specified in the call. These clock IDs can then be
978 used in the rest of the clock function calls.

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

988 **B.14.3.3 Overhead Considerations**

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

1000 **B.14.3.4 Accuracy of CPU-time Clocks**

1001 The mechanism used to measure the execution time of processes and threads is
1002 specified in this document as implementation defined. The reason for this is that
1003 both the underlying hardware and the implementation architecture have a very
1004 strong influence on the accuracy achievable for measuring CPU-time. For some
1005 implementations, the specification of strict accuracy requirements would
1006 represent very large overheads, or even the impossibility of being implemented.

1007 Since the mechanism for measuring execution time is implementation defined,
1008 realtime applications will be able to take advantage of accurate implementations
1009 using a portable interface. Of course, strictly conforming applications cannot rely
1010 on any particular degree of accuracy, in the same way as they cannot rely on a
1011 very accurate measurement of wall clock time. There will always exist applica-
1012 tions whose accuracy or efficiency requirements on the implementation are more
1013 rigid than the values defined in this or any other standard.

1014 In any case, there is a minimum set of characteristics that realtime applications
1015 would expect from most implementations. One such characteristic is that the sum
1016 of all the execution times of all the threads in a process equals the process execu-
1017 tion time, when no CPU-time clocks are disabled. This need not always be the
1018 case because implementations may differ in how they account for time during con-
1019 text switches. Another characteristic is that the sum of the execution times of all
1020 processes in a system equals the number of processors, multiplied by the elapsed
1021 time, assuming that no processor is idle during that elapsed time. However, in
1022 some systems it might not be possible to relate CPU-time to elapsed time. For

1023 example, in a heterogeneous multiprocessor system in which each processor runs
1024 at a different speed, an implementation may choose to define each “second” of
1025 CPU-time to be a certain number of “cycles” that a CPU has executed.

1026 **B.14.3.5 Existing Practice**

1027 Measuring and limiting the execution time of each concurrent activity are com-
1028 mon features of most industrial implementations of realtime systems. Almost all
1029 critical realtime systems are currently built upon a cyclic executive. With this
1030 approach, a regular timer interrupt kicks off the next sequence of computations.
1031 It also checks that the current sequence has completed. If it has not, then some
1032 error recovery action can be undertaken (or at least an overrun is avoided).
1033 Current software engineering principles and the increasing complexity of software
1034 are driving application developers to implement these systems on multi-threaded
1035 or multi-process operating systems. Therefore, if a POSIX operating system is to be
1036 used for this type of application then it must offer the same level of protection.

1037 Execution time clocks are also common in most UNIX implementations, although
1038 these clocks usually have requirements different from those of realtime applica-
1039 tions. The POSIX.1 *times()* function supports the measurement of the execution
1040 time of the calling process, and its terminated child processes. This execution time
1041 is measured in clock ticks and is supplied as two different values with the user
1042 and system execution times, respectively. BSD supports the function *getrusage()*,
1043 which allows the calling process to get information about the resources used by
1044 itself and/or all of its terminated child processes. The resource usage includes user
1045 and system CPU time. Some UNIX systems have options to specify high resolution
1046 (up to one microsecond) CPU time clocks using the *times()* or the *getrusage()* func-
1047 tions.

1048 The *times()* and *getrusage()* interfaces do not meet important realtime require-
1049 ments such as the possibility of monitoring execution time from a different process
1050 or thread, or the possibility of detecting an execution time overrun. The latter
1051 requirement is supported in some UNIX implementations that are able to send a
1052 signal when the execution time of a process has exceeded some specified value. For
1053 example, BSD defines the functions *getitimer()* and *setitimer()*, which can operate
1054 either on a realtime clock (wall-clock), or on virtual-time or profile-time clocks
1055 which measure CPU time in two different ways. These functions do not support
1056 access to the execution time of other processes. System V supports similar func-
1057 tions after release 4. Some emerging implementations of threads also support
1058 these functions.

1059 IBM’s MVS operating system supports per-process and per-thread execution time
1060 clocks. It also supports limiting the execution time of a given process.

1061 Given all this existing practice, the Working Group considered that the POSIX.1b
1062 clocks and timers interface was appropriate to meet most of the requirements that
1063 real-time applications have for execution time clocks. Functions were added to get
1064 the CPU time clock IDs, and to allow/disallow the thread CPU time clocks (in order
1065 to preserve the efficiency of some implementations of threads).

1066 **B.14.3.6 Clock Constants**

1067 The definition of the manifest constants `CLOCK_PROCESS_CPUTIME_ID` and
1068 `CLOCK_THREAD_CPUTIME_ID` allows processes or threads, respectively, to access
1069 their own execution-time clocks. However, given a process or thread, access to its
1070 own execution-time clock is also possible if the clock ID of this clock is obtained
1071 through a call to `clock_getcpuclockid()` or `pthread_getcpuclockid()`. Therefore,
1072 these constants are not necessary and could be deleted to make the interface
1073 simpler. Their existence saves one system call in the first access to the CPU-time
1074 clock of each process or thread. The Working Group considered this issue and
1075 decided to leave the constants in the standard because they are closer to the
1076 POSIX.1b use of clock identifiers.

1077 **B.14.3.7 Library Implementations of Threads**

1078 In library implementations of threads, kernel entities and library threads can
1079 coexist. In this case, if the CPU-time clocks are supported, most of the clock and
1080 timer functions will need to have two implementations: one in the thread library,
1081 and one in the system calls library. The main difference between these two imple-
1082 mentations is that the thread library implementation will have to deal with clocks
1083 and timers that reside in the thread space, while the kernel implementation will
1084 operate on timers and clocks that reside in kernel space. In the library implemen-
1085 tation, if the clock ID refers to a clock that resides in the kernel, a kernel call will
1086 have to be made. The correct version of the function can be chosen by specifying
1087 the appropriate order for the libraries during the link process.

1088 **B.14.3.8 History of Resolution Issues: Deletion of the `enable` attribute**

1089 In the draft corresponding to the first balloting round, CPU-time clocks had an
1090 attribute called `enable`. This attribute was introduced by the Working Group to
1091 allow implementations to avoid the overhead of measuring execution time for
1092 those processes or threads for which this measurement was not required. How-
1093 ever, the `enable` attribute got several ballot objections. The main reason was that
1094 processes are already required to measure execution time by the POSIX.1 `times()` C
1095 function. Consequently, the `enable` attribute was considered unnecessary, and C
1096 was deleted from the draft.

1097 **B.14.4 Rationale Relating to Timeouts**

1098 **B.14.4.1 Requirements for Timeouts**

1099 Realtime systems which must operate reliably over extended periods without
1100 human intervention are characteristic in embedded applications such as avionics,
1101 machine control, and space exploration, as well as more mundane applications
1102 such as cable TV, security systems and plant automation. A multi-tasking para-
1103 digm, in which many independent and/or cooperating software functions relinqu-
1104 ish the processor(s) while waiting for a specific stimulus, resource, condition, or
1105 operation completion, is very useful in producing well engineered programs for
1106 such systems. For such systems to be robust and fault tolerant, expected
1107 occurrences that are unduly delayed or that never occur must be detected so that
1108 appropriate recovery actions may be taken. This is difficult if there is no way for a
1109 task to regain control of a processor once it has relinquished control (blocked)
1110 awaiting an occurrence which, perhaps because of corrupted code, hardware mal-
1111 function, or latent software bugs, will not happen when expected. Therefore, the
1112 common practice in realtime operating systems is to provide a capability to time
1113 out such blocking services. Although there are several methods to achieve this
1114 already defined by POSIX, none are as reliable or efficient as initiating a timeout
1115 simultaneously with initiating a blocking service. This is especially critical in
1116 hard-realtime embedded systems because the processors typically have little time
1117 reserve, and allowed fault recovery times are measured in milliseconds rather
1118 than seconds.

1119 The working group largely agreed that such timeouts were necessary and ought to
1120 become part of the standard, particularly vendors of realtime operating systems
1121 whose customers had already expressed a strong need for timeouts. There was
1122 some resistance to inclusion of timeouts in the standard because the desired
1123 effect, fault tolerance, could, in theory, be achieved using existing facilities and
1124 alternative software designs, but there was no compelling evidence that realtime
1125 system designers would embrace such designs at the sacrifice of performance
1126 and/or simplicity.

1127 **B.14.4.2 Which Services Should Be Timed Out?**

1128 Originally, the working group considered the prospect of providing timeouts on *all*
1129 blocking services, including those currently existing in POSIX.1, POSIX.1b, and
1130 POSIX.1c, and future interfaces to be defined by other working groups, as sort of a
1131 general policy. This was rather quickly rejected because of the scope of such a
1132 change, and the fact that many of those services would not normally be used in a
1133 realtime context. More traditional time-sharing solutions to timeout would suffice
1134 for most of the POSIX.1 interfaces, while others had asynchronous alternatives
1135 which, while more complex to utilize, would be adequate for some realtime and all
1136 non-realtime applications.

1137 The list of potential candidates for timeouts was narrowed to the following for
1138 further consideration:

1139 POSIX.1b

1140 — *sem_wait()*1141 — *mq_receive()*1142 — *mq_send()*1143 — *lio_listio()*1144 — *aio_suspend()*1145 — *sigwait()*1146 timeout already implemented by *sigtimedwait()*

1147 POSIX.1c

1148 — *pthread_mutex_lock()*1149 — *pthread_join()*1150 — *pthread_cond_wait()*1151 timeout already implemented by *pthread_cond_timedwait()*

1152 POSIX.1

1153 — *read()*1154 — *write()*

1155 After further review by the working group, the *read()*, *write()*, and *lio_listio()*
 1156 functions (all forms of blocking synchronous I/O) were eliminated from the list
 1157 because

1158 (1) asynchronous alternatives exist,

1159 (2) timeouts can be implemented, albeit non-portably, in device drivers, and

1160 (3) a strong desire not to introduce modifications to POSIX.1 interfaces.

1161 The working group ultimately rejected *pthread_join()* since both that interface
 1162 and a timed variant of that interface are non-minimal and may be implemented as
 1163 a library function. See B.14.4.3 for a library implementation of *pthread_join()*.

1164 Thus there was a consensus among the working group members to add timeouts
 1165 to 4 of the remaining 5 functions (the timeout for *aio_suspend()* was ultimately
 1166 added directly to POSIX.1b, while the others are added here in POSIX.1d). How-
 1167 ever, *pthread_mutex_lock()* remained contentious.

1168 Many feel that *pthread_mutex_lock()* falls into the same class as the other func-
 1169 tions; that is, it is desirable to time out a mutex lock because a mutex may fail to
 1170 be unlocked due to errant or corrupted code in a critical section (looping or
 1171 branching outside of the unlock code), and therefore is equally in need of a reli-
 1172 able, simple, and efficient timeout. In fact, since mutexes are intended to guard
 1173 small critical sections, most *pthread_mutex_lock()* calls would be expected to
 1174 obtain the lock without blocking nor utilizing any kernel service, even in imple-
 1175 mentations of threads with global contention scope; the timeout alternative need
 1176 only be considered after it is determined that the thread must block.

1177 Those opposed to timing out mutexes feel that the very simplicity of the mutex is
 1178 compromised by adding a timeout semantic, and that to do so is senseless. They
 1179 claim that if a timed mutex is really deemed useful by a particular application,
 1180 then it can be constructed from the facilities already in POSIX.1b and POSIX.1c.
 1181 The following two C language library implementations of mutex locking with
 1182 timeout represent the solutions offered (in both implementations, the timeout
 1183 parameter is specified as absolute time, not relative time as in the proposed
 1184 POSIX.1c interfaces):

```

1185 _____
1186 #include <pthread.h>
1187 #include <time.h>
1188 #include <errno.h>
1189
1189 int pthread_mutex_timedlock(pthread_mutex_t *mutex,
1190                             const struct timespec *timeout)
1191     {
1192     struct timespec timenow;
1193
1193     while (pthread_mutex_trylock(mutex) == EBUSY)
1194     {
1195     clock_gettime(CLOCK_REALTIME, &timenow);
1196     if (timespec_cmp(&timenow, timeout) >= 0)
1197     {
1198     return ETIMEDOUT;
1199     }
1200     pthread_yield();
1201     }
1202     return 0;
1203     }
1204 _____

```

1205

Figure B-4 – Spinlock Implementation

1206 The Spinlock implementation is generally unsuitable for any application using
 1207 priority based thread scheduling policies such as {SCHED_FIFO} or {SCHED_RR},
 1208 since the mutex could currently be held by a thread of lower priority within the
 1209 same allocation domain, but since the waiting thread never blocks, only threads of
 1210 equal or higher priority will ever run, and the mutex can not be unlocked. Setting
 1211 priority inheritance or priority ceiling protocol on the mutex does not solve this
 1212 problem, since the priority of a mutex owning thread is only boosted if higher
 1213 priority threads are blocked waiting for the mutex, clearly not the case for this
 1214 spinlock.

1215 The Condition Wait implementation effectively substitutes the
 1216 *pthread_cond_timedwait()* function (which is currently timed out) for the desired
 1217 *pthread_mutex_timedlock()*. Since waits on condition variables currently do not
 1218 include protocols which avoid priority inversion, this method is generally unsuit-
 1219 able for realtime applications because it does not provide the same priority inver-
 1220 sion protection as the untimed *pthread_mutex_lock()*. Also, for any given imple-
 1221 mentations of the current mutex and condition variable primitives, this library
 1222 implementation has a performance cost at least 2.5 times that of the untimed
 1223 *pthread_mutex_lock()* even in the case where the timed mutex is readily locked

Copyright © 1999 IEEE. All rights reserved.
 This is an unapproved IEEE Standards Draft, subject to change.

```

1224 _____
1225 #include <pthread.h>
1226 #include <time.h>
1227 #include <errno.h>
1228
1228 struct timed_mutex
1229     {
1230     int locked;
1231     pthread_mutex_t mutex;
1232     pthread_cond_t cond;
1233     };
1234 typedef struct timed_mutex timed_mutex_t;
1235
1235 int timed_mutex_lock(timed_mutex_t *tm,
1236                    const struct timespec *timeout)
1237     {
1238     int timedout=FALSE;
1239     int error_status;
1240
1240     pthread_mutex_lock(&tm->mutex);
1241
1241     while (tm->locked && !timedout)
1242     {
1243     if ((error_status=pthread_cond_timedwait(&tm->cond,
1244     &tm->mutex,
1245     timeout))!=0)
1246     {
1247     if (error_status==ETIMEDOUT) timedout = TRUE;
1248     }
1249     }
1250
1250     if(timedout)
1251     {
1252     pthread_mutex_unlock(&tm->mutex);
1253     return ETIMEDOUT;
1254     }
1255     else
1256     {
1257     tm->locked = TRUE;
1258     pthread_mutex_unlock(&tm->mutex);
1259     return 0;
1260     }
1261     }
1262
1262 void timed_mutex_unlock(timed_mutex_t *tm)
1263     {
1264     pthread_mutex_lock(&tm->mutex); /*for case assignment not atomic*/
1265     tm->locked = FALSE;
1266     pthread_mutex_unlock(&tm->mutex);
1267     pthread_cond_signal(&tm->cond);
1268     }
1269 _____

```

1270

Figure B-5 – Condition Wait Implementation

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

1271 without blocking (the interfaces required for this case are shown in bold). Even in
1272 uniprocessors or where assignment is atomic, at least an additional
1273 *pthread_cond_signal()* is required. *pthread_mutex_timedlock()* could be imple-
1274 mented at effectively no performance penalty in this case because the timeout
1275 parameters need only be considered after it is determined that the mutex cannot
1276 be locked immediately.

1277 Thus it has not yet been shown that the full semantics of mutex locking with
1278 timeout can be efficiently and reliably achieved using existing interfaces. Even if
1279 the existence of an acceptable library implementation were proven, it is difficult to
1280 justify why the *interface* itself should not be made portable, especially considering
1281 approval for the other four timeouts.

1282 **B.14.4.3 Rationale for Library Implementation of pthread_timedjoin**

1283 The *pthread_join()* C Language example shown in Figure B-6 demonstrates that it
1284 is possible, using existing *pthread* facilities, to construct a variety of thread which
1285 allows for joining such a thread, but which allows the *join* operation to time out.
1286 It does this by using a *pthread_cond_timedwait()* to wait for the thread to exit. A
1287 small *timed_thread* descriptor structure is used to pass parameters from the
1288 creating thread to the created thread, and from the exiting thread to the joining
1289 thread. This implementation is roughly equivalent to what a normal
1290 *pthread_join()* implementation would do, with the single change being that
1291 *pthread_cond_timedwait()* is used in place of a simple *pthread_cond_wait()*.

1292 Since it is possible to implement such a facility entirely from existing *pthread*
1293 interfaces, and with roughly equal efficiency and complexity to an implementation
1294 which would be provided directly by a *threads* implementation, it was the con-
1295 sensus of the working group members that any *pthread_timedjoin()* facility would
1296 be unnecessary, and should not be provided.

1297 **B.14.4.4 Form of the Timeout Interfaces**

1298 The working group considered a number of alternative ways to add timeouts to
1299 blocking services. At first, a system interface which would specify a one-shot or
1300 persistent timeout to be applied to subsequent blocking services invoked by the
1301 calling process or thread was considered because it allowed all blocking services to
1302 be timed out in a uniform manner with a single additional interface; this was
1303 rather quickly rejected because it could easily result in the wrong services being
1304 timed out.

1305 It was suggested that a timeout value might be specified as an attribute of the
1306 object (semaphore, mutex, message queue, etc.), but there was no consensus on
1307 this, either on a case-by-case basis or for all timeouts.

1308 Looking at the two existing timeouts for blocking services indicates that the work-
1309 ing group members favor a separate interface for the timed version of a function.
1310 However, *pthread_cond_timedwait()* utilizes an absolute timeout value while
1311 *sigtimedwait()* uses a relative timeout value. The working group members agreed
1312 that relative timeout values are appropriate where the timeout mechanism's pri-
1313 mary use was to deal with an unexpected or error situation, but they are

c

```

1314
1315 /*
1316  * Construct a thread variety entirely from existing functions
1317  * with which a join can be done, allowing the join to time out.
1318  */

1319 #include <pthread.h>
1320 #include <time.h>

1321 struct timed_thread {
1322     pthread_t t;
1323     pthread_mutex_t m;
1324     int exiting;
1325     pthread_cond_t exit_c;
1326     void *(*start_routine)(void *arg);
1327     void *arg;
1328     void *status;
1329 };

1330 typedef struct timed_thread *timed_thread_t;
1331 static pthread_key_t timed_thread_key;
1332 static pthread_once_t timed_thread_once = PTHREAD_ONCE_INIT;

1333 static void timed_thread_init()
1334 {
1335     pthread_key_create(&timed_thread_key, NULL);
1336 }

1337 static void *timed_thread_start_routine(void *args)

1338 /*
1339  * Routine to establish thread specific data value and run the actual
1340  * thread start routine which was supplied to timed_thread_create().
1341  */

1342 {
1343     timed_thread_t tt = (timed_thread_t) args;

1344     pthread_once(&timed_thread_once, timed_thread_init);
1345     pthread_setspecific(timed_thread_key, (void *)tt);
1346     timed_thread_exit((tt->start_routine)(tt->arg));
1347 }

1348 int timed_thread_create(timed_thread_t ttp, const pthread_attr_t *attr,
1349                       void *(*start_routine)(void *), void *arg)

1350 /*
1351  * Allocate a thread which can be used with timed_thread_join().
1352  */

1353 {
1354     timed_thread_t tt;
1355     int result;

1356     tt = (timed_thread_t) malloc(sizeof(struct timed_thread));
1357     pthread_mutex_init(&tt->m, NULL);
1358     tt->exiting = FALSE;
1359     pthread_cond_init(&tt->exit_c, NULL);

```

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```

1360     tt->start_routine = start_routine;
1361     tt->arg = arg;
1362     tt->status = NULL;

1363     if ((result = pthread_create(&tt->t, attr,
1364                               timed_thread_start_routine, (void *)tt)) != 0) {
1365         free(tt);
1366         return result;
1367     }

1368     pthread_detach(tt->t);
1369     ttp = tt;
1370     return 0;
1371 }

1372 timed_thread_join(timed_thread_t tt,
1373                  struct timespec *timeout,
1374                  void **status)
1375 {
1376     int result;

1377     pthread_mutex_lock(&tt->m);
1378     result = 0;
1379     /*
1380      * Wait until the thread announces that it's exiting, or until timeout.
1381      */
1382     while (result == 0 && ! tt->exiting) {
1383         result = pthread_cond_timedwait(&tt->exit_c, &tt->m, timeout);
1384     }
1385     pthread_mutex_unlock(&tt->m);
1386     if (result == 0 && tt->exiting) {
1387         *status = tt->status;
1388         free((void *)tt);
1389         return result;
1390     }
1391     return result;
1392 }

1393 timed_thread_exit(void *status)
1394 {
1395     timed_thread_t tt;
1396     void *specific;

1397     if ((specific=pthread_getspecific(timed_thread_key)) == NULL){
1398         /*
1399          * Handle cases which won't happen with correct usage.
1400          */
1401         pthread_exit(NULL);
1402     }
1403     tt = (timed_thread_t) specific;
1404     pthread_mutex_lock(&tt->m);
1405     /*
1406      * Tell a joiner that we're exiting.
1407      */
1408     tt->status = status;
1409     tt->exiting = TRUE;
1410     pthread_cond_signal(&tt->exit_c);
1411     pthread_mutex_unlock(&tt->m);

```

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```

1412     /*
1413     * Call pthread_exit() to call destructors and really exit the thread.
1414     */
1415     pthread_exit(NULL);
1416 }
1417

```

1418 **Figure B-6 – *pthread_join()* with timeout**

1419 inappropriate when the timeout must expire at a particular time, or before a D
 1420 specific deadline. For the timeouts being introduced in this document, the work- D
 1421 ing group considered allowing both relative and absolute timeouts as is done with C
 1422 POSIX.1b timers, but ultimately favored the simpler absolute timeout form. C

1423 An absolute time measure can be easily implemented on top of an interface that D
 1424 specifies relative time, by reading the clock, calculating the difference between the D
 1425 current time and the desired wake up time, and issuing a relative timeout call. D
 1426 But there is a race condition with this approach because the thread could be D
 1427 preempted after reading the clock, but before making the timed out call; in this D
 1428 case, the thread would be awakened later than it should and, thus, if the wake up D
 1429 time represented a deadline, it would miss it. D

1430 There is also a race condition when trying to build a relative timeout on top of an D
 1431 interface that specifies absolute timeouts. In this case, we would have to read the D
 1432 clock to calculate the absolute wake up time as the sum of the current time plus D
 1433 the relative timeout interval. In this case, if the thread is preempted after reading D
 1434 the clock but before making the timed out call, the thread would be awakened ear- D
 1435 lier than desired. D

1436 But the race condition with the absolute timeouts interface is not as bad as the D
 1437 one that happens with the relative timeout interface, because there are simple D
 1438 workarounds. For the absolute timeouts interface, if the timing requirement is a D
 1439 deadline, we can still meet this deadline because the thread woke up earlier than D
 1440 the deadline. If the timeout is just used as an error recovery mechanism, the pre- D
 1441 cision of timing is not really important. If the timing requirement is that between D
 1442 actions A and B a minimum interval of time must elapse, we can safely use the D
 1443 absolute timeout interface by reading the clock after action A has been started. It D
 1444 could be argued that, since the call with the absolute timeout is atomic from the D
 1445 application point of view, it is not possible to read the clock after action A, if this D
 1446 action is part of the timed out call. But if we look at the nature of the calls for D
 1447 which we specify timeouts (locking a mutex, waiting for a semaphore, waiting for a D
 1448 message, or waiting until there is space in a message queue), the timeouts that an D
 1449 application would build on these actions would not be triggered by these actions D
 1450 themselves, but by some other external action. For example, if we want to wait for D
 1451 a message to arrive to a message queue, and wait for at least 20 milliseconds, this D
 1452 time interval would start to be counted from some event that would trigger both D
 1453 the action that produces the message, as well as the action that waits for the mes- D
 1454 sage to arrive, and not by the wait-for-message operation itself. In this case, we D
 1455 could use the workaround proposed above. D

1456 For these reasons, the absolute timeout is preferred over the relative timeout
1457 interface.

1458

1459 ⇒ **Annex B Rationale and Notes** *Add the following subclause.*

1460 NOTE: When this standard is approved, the section number of this subclause will be changed to
1461 make it consistent with the base standard and all its approved amendments.

1462 **B.20 Advisory Information**

1463 The POSIX.1b standard contains an Informative Annex with proposed interfaces
1464 for "real-time files". These interfaces could determine groups of the exact parame-
1465 ters required to do "direct I/O" or "extents". These interfaces were objected to by a
1466 significant portion of the balloting group as too complex. A portable application
1467 had little chance of correctly navigating the large parameter space to match its
1468 desires to the system. In addition, they only applied to a new type of file (real-time
1469 files) and they told the implementation exactly what to do as opposed to advising
1470 the implementation on application behavior and letting it optimize for the system
1471 the (portable) application was running on. For example, it was not clear how a sys-
1472 tem that had a disk array should set its parameters.

1473 There seemed to be several overall goals:

1474 — Optimizing Sequential Access

1475 — Optimizing Caching Behavior

1476 — Optimizing I/O data transfer

1477 — Preallocation

1478 The advisory interfaces, *posix_fadvise()* and *posix_madvise()* satisfy the first two
1479 goals. The `POSIX_FADV_SEQUENTIAL` and `POSIX_MADV_SEQUENTIAL` *advice*
1480 tells the implementation to expect serial access. Typically the system will prefetch
1481 the next several serial accesses in order to overlap I/O. It may also free previously
1482 accessed serial data if memory is tight. If the application is not doing serial access
1483 it can use `POSIX_FADV_WILLNEED` and `POSIX_MADV_WILLNEED` to accomplish
1484 I/O overlap, as required. When the application advises `POSIX_FADV_RANDOM` or
1485 `POSIX_MADV_RANDOM` behavior, the implementation usually tries to fetch a
1486 minimum amount of data with each request and it does not expect much locality.
1487 `POSIX_FADV_DONTNEED` and `POSIX_MADV_DONTNEED` allow the system to free
1488 up caching resources as the data will not be required in the near future.

1489 `POSIX_FADV_NOREUSE` tells the system that caching the specified data is not
1490 optimal. For file I/O, the transfer should go directly to the user buffer instead of
1491 being cached internally by the implementation. To portably perform direct disk
1492 I/O on all systems, the application must perform its I/O transfers according to the
1493 following rules:

- 1494 (1) The user buffer should be aligned according to the {POSIX_REC_XFER_-
1495 ALIGN} *pathconf()* variable.
- 1496 (2) The number of bytes transferred in an I/O operation should be a multiple
1497 of the {POSIX_ALLOC_SIZE_MIN} *pathconf()* variable.
- 1498 (3) The offset into the file at the start of an I/O operation should be a multi-
1499 ple of the {POSIX_ALLOC_SIZE_MIN} *pathconf()* variable.
- 1500 (4) The application should ensure that all threads which open a given file
1501 specify POSIX_FADV_NOREUSE to be sure that there is no unexpected
1502 interaction between threads using buffered I/O and threads using direct
1503 I/O to the same file.

1504 In some cases, a user buffer must be properly aligned in order to be transferred
1505 directly to/from the device. The {POSIX_REC_XFER_ALIGN} *pathconf()* variable
1506 tells the application the proper alignment.

1507 The preallocation goal is met by the space control function, *posix_fallocate()*. The
1508 application can use *posix_fallocate()* to guarantee no [ENOSPC] errors and to
1509 improve performance by prepaying any overhead required for block allocation. C

1510 Implementations may use information conveyed by a previous *posix_fadvise()* call
1511 to influence the manner in which allocation is performed. For example, if an
1512 application did the following calls:

```
1513     fd = open("file")
1514     posix_fadvise(fd, offset, len, POSIX_FADV_SEQUENTIAL)
1515     posix_fallocate(fd, len, size)
```

1516 An implementation might allocate the file contiguously on disk.

1517 C

1518 Finally, the *pathconf()* variables {POSIX_REC_MIN_XFER_SIZE}, {POSIX_REC_-
1519 MAX_XFER_SIZE} and {POSIX_REC_INCR_XFER_SIZE} tell the application a range
1520 of transfer sizes that are recommended for best I/O performance.

1521 Where bounded response time is required, the vendor can supply the appropriate
1522 settings of the advisories to achieve a guaranteed performance level.

1523 The interfaces meet the goals while allowing applications using regular files to
1524 take advantage of performance optimizations. The interfaces tell the implementa-
1525 tion expected application behavior which the implementation can use to optimize
1526 performance on a particular system with a particular dynamic load.

1527 The *posix_memalign()* function was added to allow for the allocation of specifically
1528 aligned buffers, e.g. for {POSIX_REC_XFER_ALIGN}.

1529 The working group also considered the alternative of adding a function which
1530 would return an aligned pointer to memory within a user supplied buffer. This
1531 was not considered to be the best method, because it potentially wastes large
1532 amounts of memory when buffers need to be aligned on large alignment bound-
1533 aries.

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Identifier Index

<i>clock_getcpuclockid()</i>	Accessing a Process CPU-time Clock {14.3.2}	49
<i>mq_timedreceive()</i>	Receive a Message from a Message Queue {15.2.5}	55
<i>mq_timedsend()</i>	Send a Message to a Message Queue {15.2.4}	53
<i>posix_fadvise()</i>	File Advisory Information {20.1.1}	63
<i>posix_fallocate()</i>	File Space Control {20.1.2}	64
<i>posix_madvise()</i>	Memory Advisory Information {20.2.1}	66
<i>posix_memalign()</i>	Aligned Memory Allocation {20.2.2}	68
<i>posix_spawn()</i>	Spawn a Process {3.1.6}	21
<i>posix_spawnattr_destroy()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_getflags()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_getpgroup()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_getschedparam()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_getschedpolicy()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_getsigdefault()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_getsigmask()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_init()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_setflags()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_setpgroup()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_setschedparam()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_setschedpolicy()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_setsigdefault()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawnattr_setsigmask()</i>	Spawn Attributes {3.1.5}	16
<i>posix_spawn_file_actions_addclose()</i>	Spawn File Actions {3.1.4}	14
<i>posix_spawn_file_actions_adddup2()</i>	Spawn File Actions {3.1.4}	14

Copyright © 1999 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

<i>posix_spawn_file_actions_addopen()</i>	Spawn File Actions {3.1.4}.....	14
<i>posix_spawn_file_actions_destroy()</i>	Spawn File Actions {3.1.4}.....	14
<i>posix_spawn_file_actions_init()</i>	Spawn File Actions {3.1.4}.....	14
<i>posix_spawnnp()</i>	Spawn a Process {3.1.6}.....	21
<i>pthread_getcpuclockid()</i>	Accessing a Thread CPU-time Clock {14.3.3}.....	50
<i>pthread_mutex_timedlock()</i>	Locking and Unlocking a Mutex {11.3.3}.....	35
<i>sem_timedwait()</i>	Lock a Semaphore {11.2.6}.....	33
<spawn.h>	Spawn File Actions {3.1.4}.....	14

Alphabetic Topical Index

A

abbreviations
 C Standard ... 5
 Abbreviations ... 5
 abbreviations
 POSIX.1 ... 6
 POSIX.1b ... 6
 POSIX.1c ... 6
 POSIX.1d ... 6
 POSIX.5 ... 6
 Accessing a Process CPU-time Clock ... 49
 Accessing a Thread CPU-time Clock ... 50
 Accuracy of CPU-time Clocks ... 95
 address space ... 67
 Advisory Information ... 63, 106
 Advisory Information option ... 7-8, 11, 27,
 29, 63-68
ai_suspend() ... 99
alarm() ... 25
 Aligned Memory Allocation ... 68
 ARG_MAX ... 22
 Asynchronous Input and Output ... 31
 Asynchronous I/O Control Block ... 31
 attributes
 enable ... 97
 schedparam ... 43
 attributes
 cpu-clock-requirement ... 95
 spawn-default ... 19
 spawn-flags ... 18, 20, 22-25
 spawn-pgroup ... 18, 20, 22-23, 78
 spawn-schedparam ... 19-20, 23
 spawn-schedpolicy ... 19-20, 23
 spawn-sigdefault ... 19-20, 23-24
 spawn-sigmask ... 18, 20, 23

B

background ... 89-90
 background priority ... 89-90
 Bibliography ... 71
 blocked thread ... 40-41
 bounded response ... 1, 93, 107

BSD ... 96

C

cancellation point ... 61
 Cancellation Points ... 61
chmod() ... 25
 C Language Definitions ... 7
 clock
 system ... 33, 36, 54, 56
 Clock and Timer Functions — Description
 ... 47
 Clock and Timer Functions ... 47
 clock
 CLOCK_REALTIME ... 33, 36, 54, 56
 Clock Constants ... 97
 clock
 CPU-time ... 5, 13, 47-50, 59, 94-97
clock_getcpuclockid() ... 8, 47, 49-51, 94, 97
 function definition ... 49
clock_getres() ... 47, 50-51
clock_gettime() ... 47, 50-51
 CLOCK_PROCESS_CPUTIME_ID ... 47, 94, 97
 CLOCK_REALTIME ... 33, 36, 54, 56, 100
 clock resolution ... 34, 36, 54, 56
 Clocks ... 47
 CPU-time ... 13-14
 Clocks and Timers ... 47-48, 93
 clocks
 CPU-Time ... 13
 CPU-time ... 47
clock_settime() ... 41, 47, 50-51
 CLOCK_THREAD_CPUTIME_ID ... 47, 94, 97
close() ... 16, 25
 Compile-Time Symbolic Constants for Portability Specifications ... 10-11
 Condition Wait Implementation ... 101
 Configurable Pathname Variables ... 29
 Configurable System Variables ... 27
 conformance ... 3
 implementation ... 3
 Conformance ... 3

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

- Conforming Implementation Options ... 3
 - CPU ... 5-6, 13-14, 47-50, 93-97
 - cpu-clock-requirement
 - attribute ... 95
 - CPU-time Clock Characteristics ... 49
 - CPU-Time clock ... 13
 - CPU-time clock ... 13-14, 47
 - definition of ... 5
 - CPU time [execution time]
 - definition of ... 5
 - CPU-time timer
 - definition of ... 5
 - creat()* ... 65-66
 - Create a Per-Process Timer — Description ... 48
 - Create a Per-Process Timer — Errors ... 48
 - Create a Per-Process Timer ... 48
 - Cross-References ... 16, 20, 25, 34, 37, 50-51, 55, 57, 64, 66, 68-69
 - C Standard ... 5, 68
 - abbreviation ... 5
 - definition of ... 5
- ## D
- Data Definitions for Asynchronous Input and Output ... 31
 - Definitions ... 5
 - Definitions and General Requirements ... 73
 - document ... 2, 18, 22, 44, 71, 95, 105
 - dup2()* ... 16, 25
 - Dynamically Setting the Sporadic Server Policy ... 92
 - Dynamic Thread Scheduling Parameters
 - Access — Description ... 45
 - Dynamic Thread Scheduling Parameters
 - Access — Errors ... 46
 - Dynamic Thread Scheduling Parameters
 - Access ... 45
- ## E
- [EBADF] ... 16, 64-65
 - EBADF ... 84
 - EBUSY ... 100
 - [EDEADLK] ... 37
 - [EFBIG] ... 65
 - effective group ID ... 23
 - effective user ID ... 23
 - [EINTR] ... 54, 56, 65
 - [EINVAL] ... 16, 20, 24, 34, 37, 54, 56, 64-65, 67-68
 - [EIO] ... 66
 - enable
 - attribute ... 97
 - [ENODEV] ... 66
 - [ENOMEM] ... 16, 20, 67-68
 - ENOMEM ... 85
 - [ENOSPC] ... 66, 107
 - [ENOTSUP] ... 46, 48
 - [EPERM] ... 50
 - [ESPIPE] ... 64, 66
 - [ESRCH] ... 50-51
 - [ETIMEDOUT] ... 34, 37, 54, 56
 - ETIMEDOUT ... 100-101
 - exec ... 13, 24, 75-77
 - Execute a File — Description ... 13
 - Execute a File ... 13
 - Execution Scheduling ... 39, 89
 - execution time
 - definition of ... 5
 - execution time measurement ... 73
 - Execution Time Monitoring ... 48, 93
 - Execution Time Monitoring Interface ... 94
 - Existing Practice ... 91, 96
 - _exit()* ... 25
- ## F
- FALSE ... 101, 103
 - fcntl()* ... 25
 - <fcntl.h> ... 8, 64
 - FD_CLOEXEC ... 15, 22, 76, 85
 - FIFO ... 64, 66
 - File Advisory Information ... 63
 - file descriptor ... 14-16, 22, 24, 63-65, 74-76
 - Files and Directories ... 29
 - File Space Control ... 64
 - file system ... 65-66
 - pthread_join()*
 - with timeout ... 105
 - fork()* ... 2, 13, 24-25, 75-77, 79-80
 - fork handlers ... 24
 - Form of the Timeout Interfaces ... 102
 - free()* ... 68-69

ftruncate() ... 65-66

functions

clock_getcpuclockid() ... 49
mq_timedreceive() ... 55
mq_timedsend() ... 53
posix_fadvise() ... 63
posix_fallocate() ... 64
posix_madvise() ... 66
posix_memalign() ... 68
posix_spawn() ... 21
posix_spawnattr_destroy() ... 16
posix_spawnattr_getflags() ... 16
posix_spawnattr_getpgroup() ... 16
posix_spawnattr_getschedparam() ... 16
posix_spawnattr_getschedpolicy() ... 16
posix_spawnattr_getsigdefault() ... 16
posix_spawnattr_getsigmask() ... 16
posix_spawnattr_init() ... 16
posix_spawnattr_setflags() ... 16
posix_spawnattr_setpgroup() ... 16
posix_spawnattr_setschedparam() ... 16
posix_spawnattr_setschedpolicy() ... 16
posix_spawnattr_setsigdefault() ... 16
posix_spawnattr_setsigmask() ... 16
posix_spawn_file_actions_addclose()
 ... 14
posix_spawn_file_actions_adddup2()
 ... 14
posix_spawn_file_actions_addopen()
 ... 14
posix_spawn_file_actions_destroy() ... 14
posix_spawn_file_actions_init() ... 14
posix_spawnnp() ... 21
pthread_getcpuclockid() ... 50
pthread_mutex_timedlock() ... 35
sem_timedwait() ... 33

G

General ... 1
 General Concepts — measurement of execution time ... 6
 General Concepts ... 6, 73
 General Terms ... 5
 generate a signal ... 94
 Get Configurable Pathname Variables—
 Description ... 29
 Get Configurable Pathname Variables ... 29
 Get Configurable System Variables—
 Description ... 27
 Get Configurable System Variables ... 27
getitimer() ... 96

getrusage() ... 96

H

Headers and Function Prototypes ... 7
 Historical Documentation ... 71
 Historical Documentation and Introductory
 Texts ... 71
 History of Resolution Issues: Deletion of the
 enable attribute ... 97

I

IBM ... 96
 IEEE ... 6, 78
 IEEE P1003.1a ... 76
 IEEE P1003.1d ... 6
 IEEE Std 1003.1 ... 6
 Implementation Conformance ... 3
 implementation defined ... 5-6, 19, 24, 40,
 42, 44-45, 48-49, 65, 73, 95
 Input and Output Primitives ... 31
 Introduction ... 93
 I/O Advisory Information and Space Control
 ... 63
 I/O Redirection with *posix_spawn()* ... 89
 ISO/IEC 14519 ... 6, 71, 76-77
 ISO/IEC 9899:1995 ... 5
 ISO/IEC 9899 ... 5, 68
 ISO/IEC 9945-1 ... 6
 ISO/IEC 9945 ... 7

J

job control ... 76

K

kill() ... 25

L

language binding ... 75-78
 Library-Level vs. Kernel-Level Implementa-
 tion ... 92

Copyright © 1999 IEEE. All rights reserved.

This is an unapproved IEEE Standards Draft, subject to change.

Library Implementations of Threads ... 97
 Limitation of the Number of Pending Replenishments ... 93
 <limits.h> ... 8-9
lio_listio() ... 99
 Lock a Semaphore — Cross-References ... 34
 Lock a Semaphore — Description ... 33
 Lock a Semaphore — Errors ... 34
 Lock a Semaphore — Returns ... 34
 Lock a Semaphore — Synopsis ... 33
 Lock a Semaphore ... 33
 Locking and Unlocking a Mutex — Cross-References ... 37
 Locking and Unlocking a Mutex — Description ... 36
 Locking and Unlocking a Mutex — Errors ... 37
 Locking and Unlocking a Mutex — Returns ... 36
 Locking and Unlocking a Mutex — Synopsis ... 35
 Locking and Unlocking a Mutex ... 35
 login ... 80

M

malloc() ... 69
 measurement of execution time ... 6
 Memory Advisory Information ... 66
 Memory Advisory Information and Alignment Control ... 66
 Memory Mapped Files option ... 66-67
 Message Passing ... 53
 Message Passing Functions ... 53
 Message Passing option ... 7, 53, 55
 message queues ... 53, 55
 Minimum Values ... 8
mmap() ... 68
 MMU ... 75, 77
mq_open() ... 55, 57
mq_receive() ... 55, 99
mq_send() ... 53, 99
mq_timedreceive() ... 7, 55-56, 61
 function definition ... 55
mq_timedsend() ... 7, 53-54, 61
 function definition ... 53
 <mqqueue.h> ... 8

mutexes ... 36
 Mutexes ... 35
 MVS ... 96

N

NULL ... 82-86, 103-105
 Numerical Limits ... 8

O

O_NONBLOCK ... 53-56
open() ... 16, 25, 65-66
 OPEN_MAX ... 16
 Optional Configurable Pathname Variables ... 29
 Optional Configurable System Variables ... 27
 Optional Minimum Values ... 8
 Optional Pathname Variable Values ... 10
 Optional Run-Time Invariant Values (Possibly Indeterm.) ... 9
 options
 Advisory Information ... 7-8, 11, 27, 29, 63-68
 Memory Mapped Files ... 66-67
 Message Passing ... 7, 53, 55
 Prioritized Input and Output ... 31
 Process Scheduling ... 8, 18-19, 23, 25, 31, 44
 Process Sporadic Server ... 11, 27, 39-40, 42-43
 Semaphores ... 33
 Shared Memory Objects ... 66-67
 Spawn ... 7-8, 11, 14, 17, 19, 21, 27
 Threads ... 5, 7, 24, 36
 Thread Sporadic Server ... 11, 27, 39-40, 43-45
 Timeouts ... 7, 11, 27, 33, 36, 53, 55
 Timers ... 33, 54, 56
 options
 Process CPU-Time Clocks ... 8, 11, 13, 27, 47-49, 59
 Thread CPU-Time Clocks ... 7, 11, 13, 27, 49-50, 95
 Other Standards ... 71
 Overhead Considerations ... 95

P

- package
 - POSIX_Process_Primitives ... 76
- PATH**
 - variable ... 22
- pathconf()* ... 9, 107
- pathname ... 22
- Pathname Variable Values ... 9-10
- _PC_ALLOC_SIZE_MIN ... 29
 - limit definition ... 29
- _PC_REC_INCR_XFER_SIZE ... 29
 - limit definition ... 29
- _PC_REC_MAX_XFER_SIZE ... 29
 - limit definition ... 29
- _PC_REC_MIN_XFER_SIZE ... 29
 - limit definition ... 29
- _PC_REC_XFER_ALIGN ... 29
 - limit definition ... 29
- pipe ... 64, 66, 79
- popen()* ... 79
- POSIX.1 ... 6, 8, 39, 47, 59, 77-78, 96-99
 - abbreviation ... 6
 - definition of ... 6
- POSIX.1b
 - abbreviation ... 6
 - definition of ... 6
- POSIX.1c
 - abbreviation ... 6
 - definition of ... 6
- POSIX.1d
 - abbreviation ... 6
 - definition of ... 6
- POSIX.1i ... 6
- POSIX.5 ... 6, 74, 77
 - abbreviation ... 6
 - definition of ... 6
- _POSIX_ADVISORY_INFO ... 3, 11, 27, 63, 65-66, 68
- POSIX_ALLOC_SIZE_MIN ... 10, 29, 107
- _POSIX_CPUTIME ... 3, 11, 13, 27, 47-49
- POSIX_FADV_DONTNEED ... 106
- posix_fadvise()* ... 7, 61, 63-65, 68, 106-107
 - function definition ... 63
- POSIX_FADV_NOREUSE ... 106-107
- POSIX_FADV_RANDOM ... 106
- POSIX_FADV_SEQUENTIAL ... 106-107
- POSIX_FADV_WILLNEED ... 106
- posix_fallocate()* ... 7, 61, 64-65, 107
 - function definition ... 64
- POSIX_MADV_DONTNEED ... 106
- posix_madvise()* ... 7, 61, 64, 66-67, 106
 - function definition ... 66
- POSIX_MADV_RANDOM ... 106
- POSIX_MADV_SEQUENTIAL ... 106
- POSIX_MADV_WILLNEED ... 106
- _POSIX_MAPPED_FILES ... 66
- posix_memalign()* ... 8, 68, 107
 - function definition ... 68
- _POSIX_MESSAGE_PASSING ... 53, 55
- _POSIX_PRIORITIZED_IO ... 31
- _POSIX_PRIORITY_SCHEDULING ... 11, 19, 23, 25, 31, 44
- POSIX_Process_Primitives
 - package ... 76
- POSIX_REC_INCR_XFER_SIZE ... 10, 29, 107
- POSIX_REC_MAX_XFER_SIZE ... 10, 29, 107
- POSIX_REC_MIN_XFER_SIZE ... 10, 29, 107
- POSIX_REC_XFER_ALIGN ... 10, 29, 107
- _POSIX_SEMAPHORES ... 33
- _POSIX_SHARED_MEMORY_OBJECTS ... 66
- posix_spawn()* ... 7, 14-16, 18, 20-22, 24-26, 61, 73-80, 89
 - Equivalent ... 89
 - function definition ... 21
- _POSIX_SPAWN ... 3, 11, 14, 17, 19, 21, 27
- posix_spawnattr_destroy()* ... 7, 16, 18-20, 25
 - function definition ... 16
- posix_spawnattr_getflags()* ... 7, 16, 18-20, 25
 - function definition ... 16
- posix_spawnattr_getpgroup()* ... 7, 16, 18-20, 25
 - function definition ... 16
- posix_spawnattr_getschedparam()* ... 8, 16, 19-20, 25
 - function definition ... 16
- posix_spawnattr_getschedpolicy()* ... 8, 16, 19-20, 25
 - function definition ... 16
- posix_spawnattr_getsigdefault()* ... 7, 16, 19-20, 25
 - function definition ... 16
- posix_spawnattr_getsigmask()* ... 7, 16, 18-20, 25
 - function definition ... 16
- posix_spawnattr_init()* ... 7, 16-20, 25
 - function definition ... 16
- posix_spawnattr_setflags()* ... 7, 16, 18-20, 25
 - function definition ... 16

- posix_spawnattr_setpgroup()* ... 7, 16, 18-20, 25
 - function definition ... 16
- posix_spawnattr_setschedparam()* ... 8, 16, 19-20, 25
 - function definition ... 16
- posix_spawnattr_setschedpolicy()* ... 8, 16, 19-20, 25
 - function definition ... 16
- posix_spawnattr_setsigdefault()* ... 8, 16, 19-20, 25
 - function definition ... 16
- posix_spawnattr_setsigmask()* ... 7, 16, 18-20, 25
 - function definition ... 16
- posix_spawn_file_actions_addclose()* ... 7, 14-16, 25
 - function definition ... 14
- posix_spawn_file_actions_adddup2()* ... 7, 14-16, 25
 - function definition ... 14
- posix_spawn_file_actions_addopen()* ... 7, 14-16, 25
 - function definition ... 14
- posix_spawn_file_actions_destroy()* ... 7, 14-16, 25
 - function definition ... 14
- posix_spawn_file_actions_init()* ... 7, 14-16, 25
 - function definition ... 14
- posix_spawnnp()* ... 7, 14-16, 18, 20-22, 24-26, 61, 73-80, 89
 - function definition ... 21
- POSIX_SPAWN_RESETIDS ... 18, 23, 74, 80, 82
- POSIX_SPAWN_SETPGROUP ... 18, 22-23, 25, 80, 82
- POSIX_SPAWN_SETSCHEDPARAM ... 18-19, 23, 25, 80, 85
- POSIX_SPAWN_SETSCHEDULER ... 18-19, 23, 25, 80, 84
- POSIX_SPAWN_SETSIGDEF ... 18-19, 23-24, 80, 82
- POSIX_SPAWN_SETSIGMASK ... 18, 23, 80, 82
- _POSIX_SPARADIC_SERVER ... 3, 11, 27, 39-40, 42-43
- _POSIX_SS_REPL_MAX ... 8-9, 93
- _POSIX_THREAD_CPUTIME ... 3, 11, 13, 27, 47-50, 59
- _POSIX_THREAD_PRIORITY_SCHEDULING ... 11, 43
- _POSIX_THREADS ... 36
- _POSIX_THREAD_SPORADIC_SERVER ... 3, 11, 27, 39-40, 43-45
- _POSIX_TIMEOUTS ... 3, 11, 27, 33, 36, 53, 55
- _POSIX_TIMERS ... 11
- PRIO_INHERIT ... 36
- Prioritized Input and Output option ... 31
- procedure
 - Start_Process ... 75-78
 - Start_Process_Search ... 76-78
- Process CPU-Time Clocks option ... 8, 11, 13, 27, 47-49, 59
- Process Creation — Description ... 13
- Process Creation ... 13-14
- Process Creation and Execution ... 13, 73
- Process Environment ... 27
- process group ... 18, 22-23, 26, 76, 78
- process group ID ... 23, 76
- process ID ... 23-24
- Process Management ... 13
- Process Primitives ... 73
- Process Scheduling Attributes ... 43
- Process Scheduling Functions ... 42
- Process Scheduling option ... 8, 18-19, 23, 25, 31, 44
- Process Sporadic Server option ... 11, 27, 39-40, 42-43
- Process Termination ... 26
- pthread_attr_getschedparam()* ... 45
- pthread_attr_getschedpolicy()* ... 45
- pthread_attr_setschedparam()* ... 45
- pthread_attr_setschedpolicy()* ... 45
- pthread_cond_signal()* ... 102
- pthread_cond_timedwait()* ... 99-100, 102
- pthread_cond_wait()* ... 99, 102
- pthread_create()* ... 59
- pthread_getcpuclockid()* ... 7, 47, 50-51, 94, 97
 - function definition ... 50
- pthread_getschedparam()* ... 45
- <pthread.h> ... 8
- pthread_join()* ... 99, 102
- pthread_mutex_lock()* ... 36, 99-100
- pthread_mutex_timedlock()* ... 7, 35-37, 100, 102
 - function definition ... 35
- PTHREAD_ONCE_INIT ... 103

pthread_setschedparam() ... 45-46
pthread_timedjoin() ... 102

R

Range of Scheduling Priorities ... 92
read() ... 99
 Receive a Message from a Message Queue —
 Cross-References ... 57
 Receive a Message from a Message Queue —
 Description ... 55
 Receive a Message from a Message Queue —
 Errors ... 56
 Receive a Message from a Message Queue —
 Returns ... 56
 Receive a Message from a Message Queue —
 Synopsis ... 55
 Receive a Message from a Message Queue
 ... 55
 replenishment operation ... 40-41, 93
 replenishment period ... 39-41, 90, 93
 Requirements for Timeouts ... 98
 resolution
 clock ... 34, 36, 54, 56
 Run-Time Invariant Values (Possibly Indeter-
 minate) ... 8-9
 running thread ... 40-41

S

_SC_ADVISORY_INFO ... 27
 limit definition ... 27
_SC_CPU_TIME ... 27
 limit definition ... 27
 SCHED_FIFO ... 31, 35, 40, 42-44, 92, 100
sched_get_priority_max() ... 41
sched_get_priority_min() ... 41
 <sched.h> ... 39
 SCHED_OTHER ... 40, 42, 44
 schedparam
 attribute ... 43
 SCHED_RR ... 31, 35, 40, 42-44, 100
sched_setparam() ... 25, 42
sched_setscheduler() ... 25, 43
 SCHED_SPORADIC ... 31, 35, 39-46, 92
 Scheduling Allocation Domain ... 44
 Scheduling Aperiodic Activities (rationale)
 ... 90

Scheduling Documentation ... 44
 Scheduling Parameters ... 39
 Scheduling Policies ... 39
 scheduling policy ... 35, 39-40, 42-43, 45-46,
 92-93, 100
 Scope ... 1
_SC_PAGESIZE ... 66-67
_SC_SPAWN ... 27
 limit definition ... 27
_SC_SPORADIC_SERVER ... 27
 limit definition ... 27
_SC_THREAD_CPU_TIME ... 27
 limit definition ... 27
_SC_THREAD_SPORADIC_SERVER ... 27
 limit definition ... 27
_SC_TIMEOUTS ... 27
 limit definition ... 27
 Semaphore Functions ... 33
 <semaphore.h> ... 8
 semaphores ... 33
 Semaphores option ... 33
sem_post() ... 33
sem_timedwait() ... 7, 33-34, 61
 function definition ... 33
sem_wait() ... 33, 99
 Send a Message to a Message Queue — Cross-
 References ... 55
 Send a Message to a Message Queue —
 Description ... 53
 Send a Message to a Message Queue — Errors
 ... 54
 Send a Message to a Message Queue —
 Returns ... 54
 Send a Message to a Message Queue —
 Synopsis ... 53
 Send a Message to a Message Queue ... 53
setitimer() ... 96
setpgid() ... 25, 78
 Set Scheduling Parameters — Description
 ... 42
 Set Scheduling Parameters ... 42
 Set Scheduling Policy and Scheduling Parame-
 ters — Description ... 43
 Set Scheduling Policy and Scheduling Parame-
 ters ... 43
setuid() ... 25
 Shared Memory Objects option ... 66-67
 shell ... 80
 shell ... 80

SIG_DFL ... 23, 83
 SIG_IGN ... 24
 signal
 generate ... 94
 signal actions ... 23
 signal mask ... 23
 SIG_SETMASK ... 82
sigtimedwait() ... 99, 102
sigwait() ... 99
 Spawn a Process ... 21, 75
 Spawn Attributes ... 16, 75
 spawn-default
 attribute ... 19
 Spawn File Actions ... 14, 73
 spawn-flags
 attribute ... 18, 20, 22-25
 <spawn.h> ... 8, 14, 17-18, 22
 header definition ... 14
 Spawning a new Userid Process ... 89
 spawn option ... 19
 Spawn option ... 7-8, 11, 14, 17, 21, 27
 spawn-pgroup
 attribute ... 18, 20, 22-23, 78
 spawn-schedparam
 attribute ... 19-20, 23
 spawn-schedpolicy
 attribute ... 19-20, 23
 spawn-sigdefault
 attribute ... 19-20, 23-24
 spawn-sigmask
 attribute ... 18, 20, 23
 Spinlock Implementation ... 100
 Sporadic Server Scheduling Policy ... 89
 SS_REPL_MAX ... 9, 42-43, 45, 93
 Start_Process
 procedure ... 75-78
 Start_Process_Search
 procedure ... 76-78
stat() ... 25
 <stdlib.h> ... 8
 Symbolic Constants ... 10
 Synchronization ... 33
sysconf() ... 9, 66-68
 <sys/mman.h> ... 67
system() ... 74, 76, 79
 system clock ... 33, 36, 54, 56
 System V ... 27, 96

T

Terminology and General Requirements ... 5
 terms ... 5
 Thread Cancellation ... 61
 Thread Cancellation Overview ... 61
 Thread CPU-Time Clocks option ... 7, 11, 13, 27, 49-50, 95
 Thread Creation — Description ... 59
 Thread Creation ... 59
 Thread Creation Scheduling Attributes — Description ... 45
 Thread Creation Scheduling Attributes ... 44
 Thread Management ... 59
 Threads ... 59
 Thread Scheduling ... 43
 Thread Scheduling Attributes ... 43
 Thread Scheduling Functions ... 44
 Threads option ... 5, 7, 24, 36
 Thread Sporadic Server option ... 11, 27, 39-40, 43-45
time() ... 33-34, 36-37, 54-57
 <time.h> ... 8, 34, 36-37, 49, 54-57
 Timeouts option ... 7, 11, 27, 33, 36, 53, 55
timer_create() ... 48, 50-51
 Timers option ... 33, 54, 56
times() ... 25, 96-97
 TOC ... 5
 TRUE ... 101, 104

U

undefined ... 15, 18, 40
 UNIX ... 76-77, 96
unlink() ... 66
 Unlock a Semaphore — Description ... 35
 Unlock a Semaphore ... 35
 unspecified ... 18-19, 24, 26, 31

V

Versioned Compile-Time Symbolic Constants ... 11

W

wait() ... 26, 78-79

Wait for Process Termination — Description
... 26

Wait for Process Termination ... 26

waitpid() ... 26, 78-79

WEXITSTATUS ... 79

Which Services Should Be Timed Out?
... 98

WIFEXITED ... 26, 79

WIFSIGNALED ... 26

WIFSPAWNFAIL ... 79

WIFSTOPPED ... 26

write() ... 99

WSPAWNERRNO ... 79

WSTOPSIG ... 26

