

IEEE Std 1003.1d-1999

(Amendment to
IEEE Std 1003.1-1990)

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

Sponsor

**Portable Application Standards Committee
of the
IEEE Computer Society**

Approved 16 September 1999

IEEE-SA Standards Board

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

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

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

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

Copyright © 2000 by the Institute of Electrical and Electronics Engineers, Inc.
All rights reserved. Published 16 June 2000. Printed in the United States of America.

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

No part of this publication may be reproduced in any form, in an electronic retrieval system or otherwise, without the prior written permission of the publisher.

IEEE Standards documents are developed within the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Association (IEEE-SA) Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, IEEE and the members of its societies and Standards Coordinating Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

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

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents for which a license may be required by an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

IEEE is the sole entity that may authorize the use of certification marks, trademarks, or other designations to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use is granted by the Institute of Electrical and Electronics Engineers, Inc., provided that the appropriate fee is paid to Copyright Clearance Center. To arrange for payment of licensing fee, please contact Copyright Clearance Center, Customer Service, 222 Rosewood Drive, Danvers, MA 01923 USA; (978) 750-8400. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained through the Copyright Clearance Center.

Contents

	PAGE
Introduction	v
Section 1: General	1
1.1 Scope	1
1.3 Conformance	2
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 Primitives	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	20
3.2 Process Termination	25
3.2.1 Wait for Process Termination	25
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
Section 13: Execution Scheduling	39
13.1 Scheduling Parameters	39

13.2	Scheduling Policies	39
13.2.3	SCHED_OTHER	39
13.2.4	SCHED_SPORADIC	40
13.3	Process Scheduling Functions	41
13.3.1	Set Scheduling Parameters	41
13.3.3	Set Scheduling Policy and Scheduling Parameters	42
13.4	Thread Scheduling	43
13.4.1	Thread Scheduling Attributes	43
13.4.3	Scheduling Allocation Domain	43
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	48
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
16.2	Thread Functions	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 19:	Advisory Information	63
19.1	I/O Advisory Information and Space Control	63
19.1.1	File Advisory Information	63
19.1.2	File Space Control	64
19.2	Memory Advisory Information and Alignment Control	66
19.2.1	Memory Advisory Information	66
19.2.2	Aligned Memory Allocation	67
Annex A (informative)	Bibliography	69
A.2	Other Standards	69
A.3	Historical Documentation and Introductory Texts	69
Annex B (informative)	Rationale and Notes	71
B.2	Definitions and General Requirements	71
B.3	Process Primitives	71
B.13	Execution Scheduling	87

B.14 Clocks and Timers	91
B.19 Advisory Information	102
Identifier Index	105
Alphabetic Topical Index	107

FIGURES

Figure B-1 – <i>posix_spawn()</i> Equivalent	86
Figure B-2 – I/O Redirection with <i>posix_spawn()</i>	86
Figure B-3 – Spawning a new Userid Process	87
Figure B-4 – Spinlock Implementation	97
Figure B-5 – Condition Wait Implementation	98
Figure B-6 – <i>pthread_join()</i> with timeout	101

TABLES

Table 2-3a – Optional Minimum Values	8
Table 2-5a – Optional Run-Time Invariant Values	9
Table 2-6a – Optional Pathname Variable Values	10
Table 2-10a – 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 IEEE Std 1003.1d-1999, Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application Program Interface (API)—Amendment d: Additional Realtime Extensions [C Language])

1 *Editor's Note: This introduction consists of material that will eventually be integrated into the base*
2 *POSIX.1¹⁾ standard's introduction (and the portion of Annex B that contains general rationale about*
3 *the standard). The introduction contains text that was previously held in either the foreword or*
4 *scope. As this portion of the standard is for information only, specific details of the integration*
5 *with POSIX.1 are left as an editorial exercise. The section and subclause structure of this document*
6 *follows that of POSIX.1. Sections that are not amended by this standard are omitted.*

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

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

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

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

20 The key elements of defining the scope are

- 21 (1) defining a sufficient set of functionality to cover the realtime application program
22 domain in the areas not covered by POSIX.1b and POSIX.1c;
- 23 (2) defining sufficient performance constraints and performance-related functions to
24 allow a realtime application to achieve deterministic response from the system;
25 and
- 26 (3) specifying changes or additions to improve or complete the definition of the facili-
27 ties specified in the previous real-time or threads extensions covered by POSIX.1b
28 and POSIX.1c.

29 Wherever possible, the requirements of other application environments were included in
30 the interface definition. The specific areas are noted in the scope overviews of each of the
31 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 process in
34 an efficient manner.
- 35 — Timeouts for some blocking services: additional services that provide a timeout
36 capability to system services already defined in POSIX.1b and POSIX.1c, thus

37

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

- 39 allowing the application to include better error detection and recovery capabilities.
- 40 — Sporadic server scheduling: the addition of a new scheduling policy appropriate for
41 scheduling aperiodic processes or threads in hard realtime applications.
- 42 — Execution time clocks and timers: the addition of new clocks that measure the exe-
43 cution times of processes or threads, and the possibility to create timers based upon
44 these clocks, for runtime detection (and treatment) of execution time overruns.
- 45 — Advisory information for file management: addition of services that allow the appli-
46 cation to specify advisory information that can be used by the system to achieve
47 better or even deterministic response times in file management or input and output
48 (I/O) operations.

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

- 51 — Device control: a new service to pass control information and commands between
52 the application and device drivers.
- 53 — Interrupt control: new services that allow the application to directly handle
54 hardware interrupts.

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

58 **Related Standards Activities**

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

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

- 63 (1) Additional system application program interfaces (APIs) in C language
- 64 (2) Ada and FORTRAN language bindings to (1)
- 65 (3) Shell and utility facilities
- 66 (4) Verification testing methods
- 67 (5) Realtime facilities
- 68 (6) Tracing facilities
- 69 (7) Fault tolerance
- 70 (8) Checkpoint/restart facilities
- 71 (9) Resource limiting facilities
- 72 (10) Network interface facilities
- 73 (11) System administration

74

75 2) A *Standards Status Report* that lists all current IEEE Computer Society standards projects is available from
76 the IEEE Computer Society, 1730 Massachusetts Avenue NW, Washington, DC 20036-1903; Telephone:
77 +1 202 371-0101; FAX: +1 202 728-9614. Working drafts of POSIX standards under development are available
78 from the Institute of Electrical and Electronics Engineers, 445 Hoes Lane, P.O. Box 1331, Piscataway, NJ
79 08855-1331 (<http://www.standards.ieee.org/>).

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

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

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

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

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

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

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

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

104 **Portable Application Standards Committee**

105 Chair: Lowell Johnson
106 Vice Chair: Joseph M. Gwinn
107 Functional Vice Chairs: Jay Ashford
108 Andrew Josey
109 Curtis Royster Jr.
110 Secretary: Nick Stoughton

111 **IEEE System Services Working Group—Realtime**

112 Chair: Joseph M. Gwinn
113 Susan Corwin (to 1995)
114 Secretary: Karen D. Gordon
115 Franklin C. Prindle (1996)
116 Lee Schemerhorn (to 1995)
117 Editor: Michael González
118 Robert D. Luken (to 1997)
119 Technical reviewers: Steve Brosky
120 Peter Dibble
121 Christoph Eck
122 Michael González
123 Karen D. Gordon
124 Joseph M. Gwinn
125 Franklin C. Prindle
126 Ballot coordinators: James T. Oblinger
127 Duane Hughes (to 1996)

128 **Working Group**

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

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

147	Phillip R. Acuff	Michael González	Diane Paul
148	Alejandro Alonso-Muñoz	Karen D. Gordon	Charles Pfleeger
149	Pierre-Jean Arcos	Mars J. Gralia	John Pijanowski
150	Jay Ashford	Joseph M. Gwinn	Franklin C. Prindle
151	Theodore P. Baker	Steven A. Haaser	Juan Antonio de la Puente
152	Robert Barned	Chris J. Harding	François Riche
153	Barbara K. Beauchamp	Barry Hedquist	Chuck Roark
154	Keith Bierman	Karl Heubaum	Hyman Rosen
155	Nawaf Bitar	Andrew R. Huber	Helmut Roth
156	David Black	Duane Hughes	Curtis Royster
157	David J. Blackwood	Petr Janecek	Richard Scalzo
158	Shirley Bockstahler-Brandt	Lowell G. Johnson	Richard Seibel
159	Mark Brown	Michael B. Jones	Keith Shillington
160	Alan Burns	Andrew Josey	W. Olin Sibert
161	Gregory Bussiere	Michael J. Karels	Jacob Slonim
162	H. L. Catala	James J. Keys	Nicholas M. Stoughton
163	Andrew B. Cheese	Martin J. Kirk	Gregory Swain
164	Michael W. Condry	Thomas M. Kurihara	Efstathios D. Sykas
165	Donald Cragun	Mark Larsen	Donn S. Terry
166	John S. Davies	Martin Leisner	Mark-Rene Uchida
167	Richard P. Draves	Bruce Lewis	Michael W. Vannier
168	Christoph Eck	C. Douglass Locke	Charlotte Wales
169	Philip H. Enslow	Roger J. Martin	Frederick N. Webb
170	W. Douglas Findley, Jr.	Finnbarr P. Murphy	Laurence Wolfe
171	Bill Gallmeister	Richard E. Neese	Oren Yuen
172	Michel P. Gien	James T. Oblinger	Ming De Zhou

173 The following organizational representatives voted on this standard:

174	James T. Oblinger	Diane Paul	Andrew Josey
175	<i>NGCR OSSWG</i>	<i>SAE</i>	<i>X/Open Co. Ltd.</i>

176 When the IEEE-SA Standards Board approved this standard on 16 September
177 1999, it had the following membership:

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

181	Satish K. Aggarwal	James H. Gurney	Louis-François Pau
182	Dennis Bodson	Lowell G. Johnson	Ronald C. Petersen
183	Mark D. Bowman	Robert Kennelly	Gerald H. Peterson
184	James T. Carlo	E.G. "Al" Kiener	John B. Posey
185	Gary R. Engmann	Joseph L. Koepfinger*	Gary S. Robinson
186	Harold E. Epstein	L. Bruce McClung	Akio Tojo
187	Jay Forster*	Daleep C. Mohla	Hans E. Weinrich
188	Ruben D. Garzon	Robert F. Munzner	Donald W. Zispe

189 *Member emeritus

190 Also included is the following nonvoting IEEE-SA Standards Board liaison:
191 Robert E. Hebner

192 Yvette Ho Sang
193 *IEEE Standards Project Editor*

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

Section 1: General

1 1.1 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 that 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
12 provide a required level of service in a bounded response time.

13 The key elements of defining the scope are

- 14 (1) defining a sufficient set of functionality to cover a significant part of the
15 realtime application programming domain, and
- 16 (2) defining sufficient performance constraints and performance related func-
17 tions to allow a realtime application to achieve deterministic response
18 from the system.

19 Specifically within the scope is to define interfaces that do not preclude high per-
 20 formance implementations on traditional uniprocessor realtime systems. Where-
 21 ever possible, the requirements of other application environments were included
 22 in the interface definition. The specific functional areas included in this document
 23 and their scope include

- 24 — Spawn: A process creation primitive useful for systems that have difficulty
 25 with *fork()* and as an efficient replacement for *fork()/ exec*.
- 26 — Timeouts: Alternatives to blocking primitives that provide a timeout
 27 parameter to be specified.
- 28 — Execution time monitoring: A set of execution time monitoring primitives
 29 that allow on-line measuring of thread and process execution times.
- 30 — Sporadic server: A scheduling policy for threads and processes that reserves
 31 a certain amount of execution capacity for processing aperiodic events at a
 32 given priority level.
- 33 — Advisory information: An interface that advises the implementation on
 34 (portable) application behavior so that it can optimize the system.

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

- 37 — Device control: A portable application interface to nonportable special
 38 devices.
- 39 — Interrupt control: An interface that allows a process or thread to capture an
 40 interrupt, to block awaiting the arrival of an interrupt, and to protect criti-
 41 cal sections of code that are contended for by a user-written interrupt ser-
 42 vice routine.

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

46 **1.3 Conformance**

47 **1.3.1 Implementation Conformance**

48 **1.3.1.3 Conforming Implementation Options**

49 ⇒ **1.3.1.3 Conforming Implementation Options** *Add the following to the*
 50 *table of implementation options that warrant requirement by applications or in*
 51 *specifications:*

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

Section 2: Terminology and General Requirements

1 **2.2 Definitions**

2 **2.2.2 General Terms**

3 ⇒ **2.2.2 General Terms** *Modify the contents of 2.2.2 to add the following*
 4 *definitions in the correct sorted order (disregarding the subclause numbers*
 5 *shown here).*

6 **2.2.2.1 CPU time [execution time]:** The time spent executing a process or
 7 thread, including the time spent executing system services on behalf of that pro-
 8 cess or thread. If the Threads option is supported, then the value of the CPU-time
 9 clock for a process is implementation defined. With this definition the sum of all
 10 the execution times of all the threads in a process might not equal the process exe-
 11 cution time, even in a single-threaded process, because implementations may
 12 differ in how they account for time during context switches or for other reasons.

13 **2.2.2.2 CPU-time clock:** A clock that measures the execution time of a particu-
 14 lar process or thread.

15 **2.2.2.3 CPU-time timer:** A timer attached to a CPU-time clock.

16 **2.2.2.4 execution time:** See *CPU time* in 2.2.2.1.

17 **2.2.3 Abbreviations**

18 For this standard, the following abbreviations apply:

19 **2.2.3.1 C Standard:** ISO/IEC 9899: 1995, *Information technology—Programming*
 20 *languages—C.*

21 **2.2.3.2 POSIX.1:** ISO/IEC 9945-1: 1996, (IEEE Std 1003.1-1996), *Information*
 22 *Technology—Portable Operating System Interface (POSIX®)—Part 1: System*
 23 *Application Program Interface (API) [C Language].*

24 **2.2.3.3 POSIX.1b:** IEEE Std 1003.1b-1993, *Information Technology—Portable*
25 *Operating System Interface (POSIX®)—Part 1: System Application Program*
26 *Interface (API)—Amendment b: Realtime Extensions [C Language]*, as amended
27 *by IEEE Std 1003.1i-1995, Information Technology—Portable Operating System*
28 *Interface (POSIX®)—Part 1: System Application Program Interface (API)—*
29 *Amendment i: Technical Corrigenda to Realtime Extension [C Language].*

30 **2.2.3.4 POSIX.1c:** IEEE Std 1003.1c-1995, *Information Technology—Portable*
31 *Operating System Interface (POSIX®)—Part 1: System Application Program*
32 *Interface (API)—Amendment c: Threads Extension [C Language].*

33 **2.2.3.5 POSIX.1d:** IEEE Std 1003.1d-1999, *this standard.*

34 **2.2.3.6 POSIX.5** ISO/IEC 14519:1998 {B1}¹⁾, *POSIX® Ada Language Interfaces—*
35 *Binding for System Application Program Interfaces (API) including Realtime*
36 *Extensions.* (This standard includes IEEE Std 1003.5-1992 and IEEE Std 1003.5b-
37 1996.)

38 **2.3 General Concepts**

39 ⇒ **2.3 General Concepts—measurement of execution time:** *Add the follow-*
40 *ing subclause, in the proper order, to the existing items in 2.3:*

41 **2.3.1 measurement of execution time:** The mechanism used to measure exe-
42 *cution time shall be implementation defined. The implementation shall also*
43 *define to whom will be charged the CPU time that is consumed by interrupt*
44 *handlers and system services on behalf of the operating system. Execution or CPU*
45 *time is defined in 2.2.2.1.*

46

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

49 2.7 C Language Definitions

50 2.7.3 Headers and Function Prototypes

51 ⇒ **2.7.3 Headers and Function Prototypes** *Add the following text after the*
 52 *sentence “For other functions in this part of ISO/IEC 9945, the prototypes or*
 53 *declarations shall appear in the headers listed below.”:*

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

57 ⇒ **2.7.3 Headers and Function Prototypes** *Modify the contents of subclause*
 58 *2.7.3 to add the following optional headers and functions, at the end of the*
 59 *current list of headers and functions.*

60 If the Advisory Information option is supported:

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

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

63 <mqqueue.h> *mq_timedsend(), mq_timedreceive()*

64 If the Thread CPU-Time Clocks option is supported:

65 <pthread.h> *pthread_getcpuclockid()*

66 If the Threads option and the Timeouts option are supported:

67 <pthread.h> *pthread_mutex_timedlock()*

68 If the Semaphores option and the Timeouts option are supported:

69 <semaphore.h> *sem_timedwait()*

70 If the Spawn option is supported:

71 <spawn.h> *posix_spawn(), posix_spawnnp(),*
 72 *posix_spawn_file_actions_init(),*
 73 *posix_spawn_file_actions_destroy(),*
 74 *posix_spawn_file_actions_addclose(),*
 75 *posix_spawn_file_actions_adddup2(),*
 76 *posix_spawn_file_actions_addopen(),*
 77 *posix_spawnattr_init(), posix_spawnattr_destroy(),*
 78 *posix_spawnattr_getflags(), posix_spawnattr_setflags(),*
 79 *posix_spawnattr_getpgroup(),*
 80 *posix_spawnattr_setpgroup(),*
 81 *posix_spawnattr_getsigmask(),*
 82 *posix_spawnattr_setsigmask(),*
 83 *posix_spawnattr_getsigdefault(),*
 84 *posix_spawnattr_setsigdefault()*

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

86 <spawn.h> *posix_spawnattr_getschedpolicy()*,
 87 *posix_spawnattr_setschedpolicy()*,
 88 *posix_spawnattr_getschedparam()*,
 89 *posix_spawnattr_setschedparam()*

90 If the Advisory Information option is supported:

91 <stdlib.h> *posix_memalign()*

92 If the Process CPU-Time Clocks option is supported:

93 <time.h> *clock_getcpuclockid()*

94 **2.8 Numerical Limits**

95 **2.8.2 Minimum Values**

96 ⇒ **2.8.2 Minimum Values** *Add the following text after the sentence starting*
 97 *“The symbols in Table 2-3 shall be defined in...”*

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

100 ⇒ **2.8.2 Minimum Values** *Add Table 2-3a, described below, after Table 2-3.*

101 **Table 2-3a – Optional Minimum Values**

102 Name	Description	Value	Option
103 { _POSIX_SS_REPL_MAX }	104 The number of replenishment 105 operations that may be 106 simultaneously pending for 107 a particular sporadic server scheduler.	4	Process Sporadic Server or Thread Sporadic Server

108 **2.8.4 Run-Time Invariant Values (Possibly Indeterminate)**

109 ⇒ **2.8.4 Run-Time Invariant Values (Possibly Indeterminate)** *Replace the*
 110 *whole subclause by the following text:*

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

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

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

123 ⇒ **2.8.4 Run-Time Invariant Values (Possibly Indeterminate)** *Add*
 124 *Table 2-5a, described next, after Table 2-5.*

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

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

135 **2.8.5 Pathname Variable Values**

136 ⇒ **2.8.5 Pathname Variable Values** *Replace the reference to Table 2-6 in the*
 137 *first paragraph of this subclause by:*

138 Table 2-6 or Table 2-6a

139 ⇒ **2.8.5 Pathname Variable Values** *Replace the sentence “The actual value*
 140 *supported for a specific pathname shall be provided by the *pathconf()* function”*
 141 *with the following text:*

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

146 ⇒ **2.8.5 Pathname Variable Values** *Add Table 2-6a, described next, after*
 147 *Table 2-6.*

Table 2-6a – Optional Pathname Variable Values

Name	Description	Minimum Values	Option
{POSIX_REC_INCR_XFER_SIZE}	Recommended increment for file transfer sizes between the {POSIX_REC_MIN_XFER_SIZE} and {POSIX_REC_MAX_XFER_SIZE} values.	<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

166 2.9 Symbolic Constants

167 2.9.3 Compile-Time Symbolic Constants for Portability Specifications

168 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**
 169 *Change the first words in the first paragraph, currently saying “The constants*
 170 *in Table 2-10 may be used...” to the following:*

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

172 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**
 173 *Add the following sentence at the end of the first paragraph:*

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

177 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**
 178 *Add Table 2-10a, shown below, after Table 2-10.*

179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201

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

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

202 ⇒ **2.9.3 Compile-Time Symbolic Constants for Portability Specifications**
203 *Add the following paragraphs before the last paragraph in 2.9.3:*

204 If the symbol {_POSIX_SPORADIC_SERVER} is defined, then the symbol
205 {_POSIX_PRIORITY_SCHEDULING} shall also be defined. If the symbol
206 {_POSIX_THREAD_SPORADIC_SERVER} is defined, then the symbol {_POSIX_
207 THREAD_PRIORITY_SCHEDULING} shall also be defined.

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

Section 3: Process Primitives

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 the*
5 *description of the fork() function:*

6 If `{_POSIX_CPUTIME}` is defined:

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:

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

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:

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

22 If `{_POSIX_THREAD_CPUTIME}` is defined:

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

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

26 **3.1.4 Spawn File Actions**

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

30 **3.1.4.1 Synopsis**

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

33 int posix_spawn_file_actions_init(
34     posix_spawn_file_actions_t *file_actions);

35 int posix_spawn_file_actions_destroy(
36     posix_spawn_file_actions_t *file_actions);

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

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

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

47 **3.1.4.2 Description**

48 If `{_POSIX_SPAWN}` is defined:

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

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

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

60 The *posix_spawn_file_actions_destroy()* function destroys the object refer-
 61 enced by *file_actions*; the object becomes, in effect, uninitialized. An imple-
 62 mentation may cause *posix_spawn_file_actions_destroy()* to set the object
 63 referenced by *file_actions* to an invalid value. A destroyed spawn file actions

64 object can be reinitialized using *posix_spawn_file_actions_init()*; the results
 65 of otherwise referencing the object after it has been destroyed are
 66 undefined.

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

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

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

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

90 Otherwise:

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

96 3.1.4.3 Returns

97 Upon successful completion, the *posix_spawn_file_actions_init()*,
 98 *posix_spawn_file_actions_destroy()*, *posix_spawn_file_actions_addclose()*,
 99 *posix_spawn_file_actions_adddup2()*, or *posix_spawn_file_actions_addopen()*
 100 operation shall return zero. Otherwise, an error number shall be returned to indi-
 101 cate the error.

102 3.1.4.4 Errors

103 For each of the following conditions, if the condition is detected, the
 104 *posix_spawn_file_actions_init()*, *posix_spawn_file_actions_addclose()*,
 105 *posix_spawn_file_actions_adddup2()*, or *posix_spawn_file_actions_addopen()* func-
 106 tion shall return the corresponding error number:

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

109 For each of the following conditions, if the condition is detected, the
110 *posix_spawn_file_actions_destroy()*, *posix_spawn_file_actions_addclose()*,
111 *posix_spawn_file_actions_adddup2()*, or *posix_spawn_file_actions_addopen()* func-
112 tion shall return the corresponding error number:

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

114 For each of the following conditions, the *posix_spawn_file_actions_addclose()*,
115 *posix_spawn_file_actions_adddup2()*, or *posix_spawn_file_actions_addopen()* func-
116 tion shall return the corresponding error number:

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

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

125 3.1.4.5 Cross-References

126 *close()*, 6.3.1; *dup2()*, 6.2.1; *open()*, 5.3.1; *posix_spawn()*, 3.1.6; *posix_spawnnp()*,
127 3.1.6;

128 3.1.5 Spawn Attributes

129 Functions: *posix_spawnattr_init()*, *posix_spawnattr_destroy()*,
130 *posix_spawnattr_getflags()*, *posix_spawnattr_setflags()*,
131 *posix_spawnattr_getpgroup()*, *posix_spawnattr_setpgroup()*,
132 *posix_spawnattr_getschedpolicy()*, *posix_spawnattr_setschedpolicy()*,
133 *posix_spawnattr_getschedparam()*, *posix_spawnattr_setschedparam()*,
134 *posix_spawnattr_getsigmask()*, *posix_spawnattr_setsigmask()*,
135 *posix_spawnattr_getsigdefault()*, *posix_spawnattr_setsigdefault()*.

136 3.1.5.1 Synopsis

```

137 #include <sys/types.h>
138 #include <signal.h>
139 #include <spawn.h>
140 int posix_spawnattr_init (posix_spawnattr_t *attr);
141 int posix_spawnattr_destroy (posix_spawnattr_t *attr);
142 int posix_spawnattr_getflags (const posix_spawnattr_t *attr,
143                               short *flags);
144 int posix_spawnattr_setflags (posix_spawnattr_t *attr,
145                               short flags);
146 int posix_spawnattr_getpgroup (const posix_spawnattr_t *attr,
147                                pid_t *pgroup);
148 int posix_spawnattr_setpgroup (posix_spawnattr_t *attr,
149                                pid_t pgroup);
150 int posix_spawnattr_getsigmask (const posix_spawnattr_t *attr,
151                                 sigset_t *sigmask);
152 int posix_spawnattr_setsigmask (posix_spawnattr_t *attr,
153                                 const sigset_t *sigmask);
154 int posix_spawnattr_getsigdefault (const posix_spawnattr_t *attr,
155                                    sigset_t *sigdefault);
156 int posix_spawnattr_setsigdefault (posix_spawnattr_t *attr,
157                                    const sigset_t *sigdefault);

158 #include <sched.h>
159 int posix_spawnattr_getschedpolicy (const posix_spawnattr_t *attr,
160                                     int *schedpolicy);
161 int posix_spawnattr_setschedpolicy (posix_spawnattr_t *attr,
162                                     int schedpolicy);
163 int posix_spawnattr_getschedparam (const posix_spawnattr_t *attr,
164                                    struct sched_param *schedparam);
165 int posix_spawnattr_setschedparam (posix_spawnattr_t *attr,
166                                    const struct sched_param *schedparam);

```

167 3.1.5.2 Description

168 If `{_POSIX_SPAWN}` is defined:

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

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

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

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

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

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

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

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

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

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

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

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

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

227 Otherwise:

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

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

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

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

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

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

255 Otherwise:

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

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

262 3.1.5.3 Returns

263 If successful, the *posix_spawnattr_init()*, *posix_spawnattr_destroy()*,
 264 *posix_spawnattr_setflags()*, *posix_spawnattr_setpgroup()*,
 265 *posix_spawnattr_setschedpolicy()*, *posix_spawnattr_setschedparam()*,

266 *posix_spawnattr_setsigmask()*, and *posix_spawnattr_setsigdefault()* functions
 267 shall return zero. Otherwise, an error number shall be returned to indicate the
 268 error.

269 If successful, the *posix_spawnattr_getflags()*, *posix_spawnattr_getpgroup()*,
 270 *posix_spawnattr_getschedpolicy()*, *posix_spawnattr_getschedparam()*,
 271 *posix_spawnattr_getsigmask()*, and *posix_spawnattr_getsigdefault()* functions
 272 shall return zero and store the value of the *spawn-flags*, *spawn-pgroup*,
 273 *spawn-schedpolicy*, *spawn-schedparam*, *spawn-sigmask*, or *spawn-*
 274 *sigdefault* attribute of *attr* into the object referenced by the *flags*, *pgroup*,
 275 *schedpolicy*, *schedparam*, *sigmask*, or *sigdefault* parameter, respectively. Other-
 276 wise, an error number shall be returned to indicate the error.

277 **3.1.5.4 Errors**

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

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

282 For each of the following conditions, if the condition is detected, the
 283 *posix_spawnattr_destroy()*, *posix_spawnattr_getflags()*,
 284 *posix_spawnattr_setflags()*, *posix_spawnattr_getpgroup()*,
 285 *posix_spawnattr_setpgroup()*, *posix_spawnattr_getschedpolicy()*,
 286 *posix_spawnattr_setschedpolicy()*, *posix_spawnattr_getschedparam()*,
 287 *posix_spawnattr_setschedparam()*, *posix_spawnattr_getsigmask()*,
 288 *posix_spawnattr_setsigmask()*, *posix_spawnattr_getsigdefault()*, and
 289 *posix_spawnattr_setsigdefault()* functions shall return the corresponding error
 290 value:

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

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

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

298 **3.1.5.5 Cross-References**

299 *posix_spawn()*, 3.1.6; *posix_spawnnp()*, 3.1.6.

300 **3.1.6 Spawn a Process**

301 Functions: *posix_spawn()*, *posix_spawnnp()*.

302 3.1.6.1 Synopsis

```

303 #include <sys/types.h>
304 #include <spawn.h>

305 int posix_spawn(pid_t *pid,
306                const char *path,
307                const posix_spawn_file_actions_t *file_actions,
308                const posix_spawnattr_t *attrp,
309                char * const argv[],
310                char * const envp[]);

311 int posix_spawnp(pid_t *pid,
312                 const char *file,
313                 const posix_spawn_file_actions_t *file_actions,
314                 const posix_spawnattr_t *attrp,
315                 char * const argv[],
316                 char * const envp[]);

```

317 3.1.6.2 Description

318 If `{_POSIX_SPAWN}` is defined:

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

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

```

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

```

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

```

327     extern char **environ;

```

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

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

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

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

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

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

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

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

- 363 1. The set of open file descriptors for the child process shall initially be
 364 the same set as is open for the calling process. All attributes of the
 365 corresponding open file descriptions, including file locks (see 6.5.2),
 366 shall remain unchanged.
- 367 2. The signal mask and the effective user and group IDs for the child pro-
 368 cess shall be changed as specified in the attributes object referenced
 369 by *attrp*.
- 370 3. The file actions specified by the spawn file actions object shall be per-
 371 formed in the order in which they were added to the spawn file actions
 372 object.
- 373 4. Any file descriptor that has its **FD_CLOEXEC** flag set (see 6.5.2) shall
 374 be closed.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

442 Otherwise :

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

446 3.1.6.3 Returns

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

454 3.1.6.4 Errors

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

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

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

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

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

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

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

490 3.1.6.5 Cross-References

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

505 3.2 Process Termination

506 3.2.1 Wait for Process Termination

507 3.2.1.2 Wait for Process Termination — Description

508 ⇒ **3.2.1.2 Wait for Process Termination — Description** *Add the following*
 509 *paragraphs after the definition of the WSTOPSIG(stat_val) macro:*

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

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 *WIFSIGNALED(stat_val)* if a signal is sent to the parent's process group after
 518 *posix_spawn()* or *posix_spawnp()* is called.

Section 4: Process Environment

1 4.8 Configurable System Variables

2 4.8.1 Get Configurable System Variables

3 4.8.1.2 Description

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

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

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

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

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

16 **Table 4-3 – Optional Configurable System Variables**

17	18 Variable	name Value	Required Option
19	{_POSIX_SPAWN}	_SC_SPAWN	Spawn
20	{_POSIX_TIMEOUTS}	_SC_TIMEOUTS	Timeouts
21	{_POSIX_CPUTIME}	_SC_CPUTIME	Process CPU-Time Clocks
22	{_POSIX_THREAD_CPUTIME}	_SC_THREAD_CPUTIME	Thread CPU-Time Clocks
23	{_POSIX_SPARADIC_SERVER}	_SC_SPARADIC_SERVER	Process Sporadic Server
24	{_POSIX_THREAD_SPARADIC_SERVER}	_SC_THREAD_SPARADIC_SERVER	Thread Sporadic Server
25	{_POSIX_ADVISORY_INFO}	_SC_ADVISORY_INFO	Advisory Information

Section 5: Files and Directories

1 5.7 Configurable Pathname Variables

2 5.7.1 Get Configurable Pathname Variables

3 5.7.1.2 Description

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

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

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

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

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

16 **Table 5-3 – Optional Configurable Pathname Variables**

17	18 Variable	name Value	Required Option
19	{POSIX_REC_INCR_XFER_SIZE}	_PC_REC_INCR_XFER_SIZE	Advisory Information
20	{POSIX_ALLOC_SIZE_MIN}	_PC_ALLOC_SIZE_MIN	Advisory Information
21	{POSIX_REC_MAX_XFER_SIZE}	_PC_REC_MAX_XFER_SIZE	Advisory Information
22	{POSIX_REC_MIN_XFER_SIZE}	_PC_REC_MIN_XFER_SIZE	Advisory Information
23	{POSIX_REC_XFER_ALIGN}	_PC_REC_XFER_ALIGN	Advisory Information

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**

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

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

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 at the end of the list of*
4 *functions:*

5 `sem_timedwait()`.

6 11.2.6.1 Synopsis

7 ⇒ **11.2.6.1 Lock a Semaphore—Synopsis** *Add the following #include and pro-*
8 *totype at the end of the synopsis:*

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

12 11.2.6.2 Description

13 ⇒ **11.2.6.2 Lock a Semaphore—Description** *Add the following text at the end*
14 *of the description:*

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

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

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

29 which it is based. The *timespec* datatype is defined as a structure in the
30 header `<time.h>`.

31 Under no circumstance will the function fail with a timeout if the sema-
32 phore can be locked immediately. The validity of the *abs_timeout* argu-
33 ment need not be checked if the semaphore can be locked immediately.

34 Otherwise:

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

37 **11.2.6.3 Returns**

38 ⇒ **11.2.6.3 Lock a Semaphore—Returns** *Add the following function to the list*
39 *of functions:*

40 *sem_timedwait()*

41 **11.2.6.4 Errors**

42 ⇒ **11.2.6.4 Lock a Semaphore — Errors** *Make the following changes to the*
43 *discussion of error conditions:*

44 Add *sem_timedwait()* to the list of functions for both the standard error condi-
45 tions and the “if detected” error conditions.

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

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

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

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

53 **11.2.6.5 Cross-References**

54 ⇒ **11.2.6.5 Lock a Semaphore—Cross-References** *Add the following items to*
55 *the cross-references in alphabetical order:*

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

57 **11.2.7 Unlock a Semaphore**

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

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

66 **11.3 Mutexes**

67 **11.3.3 Locking and Unlocking a Mutex**

68 ⇒ **11.3.3 Locking and Unlocking a Mutex** *Add the following function at the*
 69 *end of the 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 at the end of the synopsis:*

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

77 **11.3.3.2 Description**

78 ⇒ **11.3.3.2 Locking and Unlocking a Mutex—Description** *Add the following*
 79 *text at the end of the description:*

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

81 The *pthread_mutex_timedlock()* function is called to lock the mutex
 82 object referenced by *mutex*. If the mutex is already locked, the calling
 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*
 88 passes, as measured by the clock on which timeouts are based (that is,

89 when the value of that clock equals or exceeds *abs_timeout*), or if the
 90 absolute time specified by *abs_timeout* has already been passed at the
 91 time of the call. If the Timers option is supported, the timeout is based
 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
 96 header `<time.h>`.

97 Under no circumstance will the function fail with a timeout if the mutex
 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 Otherwise:

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

108 11.3.3.3 Returns

109 ⇒ **11.3.3.3 Locking and Unlocking a Mutex—Returns** *Add the following*
 110 *function to the list of functions:*

111 *pthread_mutex_timedlock()*

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* paramete-
 119 r 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 **11.3.3.5 Cross-References**

127 ⇒ **11.3.3.5 Locking and Unlocking a Mutex—Cross-References** *Add the fol-*
128 *lowing items to the cross-references in alphabetical order:*

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

Section 13: Execution Scheduling

1 13.1 Scheduling Parameters

2 ⇒ **13.1 Scheduling Parameters** *Add the following paragraph after the first*
3 *paragraph and associated table:*

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

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

13 13.2 Scheduling Policies

14 ⇒ **13.2 Scheduling Policies** *Add the following after the unnumbered table with*
15 *the scheduling policies that shall be defined in `<sched.h>`:*

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

18	Symbol	Description
19	<code>SCHED_SPORADIC</code>	Sporadic server scheduling policy.

20 13.2.3 SCHED_OTHER

21 ⇒ **13.2.3 SCHED_OTHER** *Change the sentence beginning with “The effect of*
22 *scheduling threads with the ... ” to the following:*

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

26 ⇒ **13.2 Scheduling Policies** *Add the following subclause:*

27 **13.2.4 SCHED_SPORADIC**

28 If `{_POSIX_SPORADIC_SERVER}` is defined or `{_POSIX_THREAD_SPORADIC-`
 29 `SERVER}` is defined, the implementation shall include a scheduling policy
 30 identified by the value `SCHED_SPORADIC`.

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

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

- 49 (1) When the thread at the head of the *sched_priority* list becomes a running
 50 thread, its execution time shall be limited to at most its available execu-
 51 tion capacity, plus the resolution of the execution time clock used for this
 52 scheduling policy. This resolution shall be implementation defined.
- 53 (2) Each time the thread is inserted at the tail of the list associated with
 54 *sched_priority* (because as a blocked thread it became runnable with
 55 priority *sched_priority* or because a replenishment operation was per-
 56 formed), the time at which this operation is done is posted as the
 57 *activation_time*.
- 58 (3) When the running thread with assigned priority equal to *sched_priority*
 59 becomes a preempted thread, it becomes the head of the thread list for its
 60 priority; and the execution time consumed is subtracted from the avail-
 61 able execution capacity. If the available execution capacity would become
 62 negative by this operation, it shall be set to zero.
- 63 (4) When the running thread with assigned priority equal to *sched_priority*
 64 becomes a blocked thread, the execution time consumed is subtracted
 65 from the available execution capacity; and a replenishment operation is
 66 scheduled, as described in (6) and (7). If the available execution capacity
 67 would become negative by this operation, it shall be set to zero.
- 68 (5) When the running thread with assigned priority equal to *sched_priority*
 69 reaches the limit imposed on its execution time, it becomes the tail of the

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

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

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

92 Execution time is defined in 2.2.2.

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

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

99 13.3 Process Scheduling Functions

100 13.3.1 Set Scheduling Parameters

101 13.3.1.2 Description

102 ⇒ **13.3.1.2 Set Scheduling Parameters—Description** *Add the following*
 103 *paragraphs to the description:*

104 If `{_POSIX_SPORADIC_SERVER}` is defined:

105 If the scheduling policy of the target process is SCHED_SPORADIC, the
 106 value specified by the *sched_ss_low_priority* member of the *param* argu-
 107 ment shall be any integer within the inclusive priority range for the
 108 sporadic server policy. The *sched_ss_repl_period* and

109 *sched_ss_init_budget* members of the *param* argument shall represent
 110 the time parameters to be used by the sporadic server scheduling policy
 111 for the target process. The *sched_ss_max_repl* member of the *param*
 112 argument shall represent the maximum number of replenishments that
 113 are allowed to be pending simultaneously for the process scheduled
 114 under this scheduling policy.

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

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

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

128 ⇒ **13.3.1.2 Set Scheduling Parameters—Description** *Replace the eighth*
 129 *paragraph, beginning “If the current scheduling policy...,” with the following*
 130 *new paragraph:*

131 If the current scheduling policy for the process specified by *pid* is not
 132 SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC, the result is implementa-
 133 tion defined; this case includes the SCHED_OTHER policy.

134 **13.3.3 Set Scheduling Policy and Scheduling Parameters**

135 **13.3.3.2 Description**

136 ⇒ **13.3.3.2 Set Scheduling Policy and Scheduling Parameters—**
 137 **Description** *Add the following paragraphs to the description, before the last*
 138 *paragraph:*

139 If {_POSIX_SPORADIC_SERVER} is defined:

140 If the scheduling policy specified by *policy* is SCHED_SPORADIC, the
 141 value specified by the *sched_ss_low_priority* member of the *param* argu-
 142 ment shall be any integer within the inclusive priority range for the
 143 sporadic server policy. The *sched_ss_repl_period* and
 144 *sched_ss_init_budget* members of the *param* argument shall represent
 145 the time parameters used by the sporadic server scheduling policy for
 146 the target process. The *sched_ss_max_repl* member of the *param* argu-
 147 ment shall represent the maximum number of replenishments that are
 148 allowed to be pending simultaneously for the process scheduled under

149 this scheduling policy.

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

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

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

160 **13.4 Thread Scheduling**

161 **13.4.1 Thread Scheduling Attributes**

162 ⇒ **13.4.1 Thread Scheduling Attributes** *Add the following paragraph after*
163 *the paragraph that begins with “If the*
164 *{_POSIX_THREAD_PRIORITY_SCHEDULING} option is defined, ...”:*

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

170 **13.4.3 Scheduling Allocation Domain**

171 ⇒ **13.4.3 Scheduling Allocation Domain** *Change the first sentence of the*
172 *fourth paragraph, currently reading “For application threads whose scheduling*
173 *allocation domain size is greater than one, the rules defined for SCHED_FIFO*
174 *and SCHED_RR in 13.2 shall be used in an implementation-defined manner.” to*
175 *the following:*

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

180 ⇒ **13.4.3 Scheduling Allocation Domain**

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

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

186 **13.4.4 Scheduling Documentation**

187 ⇒ **13.4.4 Scheduling Documentation** *Change the first sentence, beginning*
 188 *with “If `{_POSIX_PRIORITY_SCHEDULING}` is defined, then ...” and ending*
 189 *with “... such a policy, are implementation defined.” to the following:*

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

195 **13.5 Thread Scheduling Functions**

196 **13.5.1 Thread Creation Scheduling Attributes**

197 **13.5.1.2 Description**

198 ⇒ **13.5.1.2 Thread Creation Scheduling Attributes—Description** *Add the*
 199 *following sentence to the sixth paragraph, beginning “The supported values of*
 200 *policy ...”:*

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

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

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

214 **13.5.2 Dynamic Thread Scheduling Parameters Access**

215 **13.5.2.2 Description**

216 ⇒ **13.5.2.2 Dynamic Thread Scheduling Parameters Access—Description**
217 *Add the following paragraph to the description, before the last paragraph:*

218 If `{_POSIX_THREAD_SPORADIC_SERVER}` is defined:

219 The *policy* argument may have the value `SCHED_SPORADIC`, with the
220 exception for the `pthread_setschedparam()` function that, if the schedul-
221 ing policy was not `SCHED_SPORADIC` at the time of the call, it is imple-
222 mentation defined whether the function is supported. In other words,
223 the implementation need not allow the application to dynamically
224 change the scheduling policy to `SCHED_SPORADIC`. The sporadic server
225 scheduling policy has the associated parameters `sched_ss_low_priority`,
226 `sched_ss_repl_period`, `sched_ss_init_budget`, `sched_priority`, and
227 `sched_ss_max_repl`. The specified `sched_ss_repl_period` shall be greater
228 than or equal to the specified `sched_ss_init_budget` for the function to
229 succeed; if it is not, then the function shall fail. The value of
230 `sched_ss_max_repl` shall be within the inclusive range `[1, {SS_REPL_-`
231 `MAX}]` for the function to succeed; if not, the function shall fail.

232 **13.5.2.4 Errors**

233 ⇒ **13.5.2.4 Dynamic Thread Scheduling Parameters Access—Errors** *Add*
234 *the following error status value in the “if detected” section of the*
235 *pthread_setschedparam() function:*

236 `[ENOTSUP]` An attempt was made to dynamically change the scheduling
237 policy to `SCHED_SPORADIC`, and the implementation does not support
238 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, after the paragraph starting “A clock may be*
 6 *systemwide...”:*

7 If `{_POSIX_CPUTIME}` is defined, implementations shall support clock ID values
 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
 15 `clock_settime()` shall have no effect on the behavior of the sporadic server
 16 scheduling policy (see 13.2.4).

17 If `{_POSIX_THREAD_CPUTIME}` is defined, implementations shall support clock
 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-
 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
 25 `clock_settime()` shall have no effect on the behavior of the sporadic server
 26 scheduling policy (see 13.2.4).

27 **14.2.2 Create a Per-Process Timer**

28 **14.2.2.2 Description**

29 ⇒ **14.2.2.2 Create a Per-Process Timer—Description** *Add the following*
 30 *paragraphs to the description, after the paragraph starting “Each implementa-*
 31 *tion shall define...”:*

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

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

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 con-*
 41 *dition:*

42 [ENOTSUP]

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

47 ⇒ **14 Clocks and Timers** *Add the following subclauses:*

48 **14.3 Execution Time Monitoring**

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

51 **14.3.1 CPU-time Clock Characteristics**

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

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

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

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

60 `CLOCK_PROCESS_CPUTIME_ID`

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

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

66 `CLOCK_THREAD_CPUTIME_ID`

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

70 **14.3.2 Accessing a Process CPU-time Clock**

71 Function: *clock_getcpuclockid()*.

72 **14.3.2.1 Synopsis**

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

74 `#include <time.h>`

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

76 **14.3.2.2 Description**

77 If `{_POSIX_CPUTIME}` is defined:

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

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

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

86 Otherwise:

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

89 **14.3.2.3 Returns**

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

92 **14.3.2.4 Errors**

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

95 [EPERM]

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

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

100 [ESRCH]

101 No process can be found corresponding to the value specified by *pid*.

102 **14.3.2.5 Cross-References**

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

105 **14.3.3 Accessing a Thread CPU-time Clock**

106 Function: *pthread_getcpuclockid()*.

107 **14.3.3.1 Synopsis**

```
108 #include <sys/types.h>
109 #include <time.h>
110 #include <pthread.h>
111 int pthread_getcpuclockid (pthread_t thread_id, clockid_t *clock_id);
```

112 **14.3.3.2 Description**

113 If `{_POSIX_THREAD_CPUTIME}` is defined:

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

117 Otherwise:

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

120 **14.3.3.3 Returns**

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

123 **14.3.3.4 Errors**

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

126 [ESRCH]

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

128 **14.3.3.5 Cross-References**

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

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 at*
 4 *the end of the list and change “Function” to “Functions”:*

5 `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:*

```
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);
```

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 at the end of the description:*

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

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()`.

26 The timeout expires when the absolute time specified by `abs_timeout`
 27 passes, as measured by the clock on which timeouts are based (that is,
 28 when the value of that clock equals or exceeds `abs_timeout`), or if the

29 absolute time specified by *abs_timeout* has already been passed at the
 30 time of the call. If the Timers option is supported, the timeout is based
 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
 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 following*
 45 *function at the end of the list and change “Function” to “Functions”:*

46 *mq_timedsend()*

47 15.2.4.4 Errors

48 ⇒ **15.2.4.4 Send a Message to a Message Queue—Errors** *Make the following*
 49 *changes to the discussion of error conditions:*

50 Add *mq_timedsend()* at the end of the list of functions to which the error condi-
 51 tions apply.

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

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

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

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

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

62 **15.2.4.5 Cross-References**63 ⇒ **15.2.4.5 Send a Message to a Message Queue—Cross-References**

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

65 `mq_open()`, 15.2.1; `time()` 4.5.1; `<time.h>`, 14.1.66 **15.2.5 Receive a Message from a Message Queue**67 ⇒ **15.2.5 Receive a Message from a Message Queue** *Add the following func-*
68 *tion at the end of the list and change “Function ” to “Functions ”:*69 `mq_timedreceive()`70 **15.2.5.1 Synopsis**71 ⇒ **15.2.5.1 Receive a Message from a Message Queue—Synopsis**72 *Add the following #include and prototype to the end of the synopsis:*73 `#include <time.h>`
74 `int mq_timedreceive(mqd_t mqdes,`
75 `char *msg_ptr,`
76 `size_t msg_len,`
77 `unsigned int *msg_prio,`
78 `const struct timespec *abs_timeout);`79 **15.2.5.2 Description**80 ⇒ **15.2.5.2 Receive a Message from a Message Queue—Description** *Add*
81 *the following text at the end of the description:*82 If `{_POSIX_MESSAGE_PASSING}` and `{_POSIX_TIMEOUTS}` are both defined:83 The `mq_timedreceive()` function is used to receive the oldest of the
84 highest priority messages from the message queue specified by `mqdes` as
85 in the `mq_receive()` function. However, if `O_NONBLOCK` was not
86 specified when the message queue was opened via the `mq_open()` func-
87 tion and no message exists on the queue to satisfy the receive, the wait
88 for such a message will be terminated when the specified timeout
89 expires. If `O_NONBLOCK` is set, this function shall behave identically to
90 `mq_receive()`.91 The timeout expires when the absolute time specified by `abs_timeout`
92 passes, as measured by the clock on which timeouts are based (that is,
93 when the value of that clock equals or exceeds `abs_timeout`), or if the
94 absolute time specified by `abs_timeout` has already been passed at the
time of the call. If the Timers option is supported, the timeout is based

95 on the `CLOCK_REALTIME` clock; if the Timers option is not supported,
 96 the timeout is based on the system clock as returned by the `time()` func-
 97 tion. The resolution of the timeout is the resolution of the clock on
 98 which it is based. The `timespec` argument is defined as a structure in
 99 the header `<time.h>`.

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

104 Otherwise:

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

107 **15.2.5.3 Returns**

108 ⇒ **15.2.5.3 Receive a Message from a Message Queue—Returns** *Add the*
 109 *following function to the list of functions:*

110 `mq_timedreceive()`

111 **15.2.5.4 Errors**

112 ⇒ **15.2.5.4 Receive a Message from a Message Queue—Errors** *Make the fol-*
 113 *lowing changes to the discussion of error conditions:*

114 Add `mq_timedreceive()` at the end of the list of functions for both the “if
 115 occurs” error conditions and the “if detected” error conditions.

116 Add an [ETIMEDOUT] error value to the “if occurs” error conditions (in alpha-
 117 betical order), with the following reason:

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

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

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

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

127 **15.2.5.5 Cross-References**

128 ⇒ **15.2.5.5 Receive a Message from a Message Queue—Cross-References**
129 *Add the following cross-references in alphabetical order:*

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

Section 16: Thread Management

1 **16.2 Thread Functions**

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 the*
5 *description, after the paragraph starting “The signal state of the new*
6 *thread...”:*

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

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 (in alphabetical*
 4 *order) to the list of functions for which a cancellation point shall occur:*

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

6 ⇒ **18.1.2 Cancellation Points** *Add the following functions (in alphabetical*
 7 *order) to the list of functions for which a cancellation point may also occur:*

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

Section 19: Advisory Information

1 ⇒ **19 Advisory Information** *Add the following section:*

2 **19.1 I/O Advisory Information and Space Control**

3 **19.1.1 File Advisory Information**

4 Function: *posix_fadvise()*.

5 **19.1.1.1 Synopsis**

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

10 **19.1.1.2 Description**

11 If `{_POSIX_ADVISORY_INFO}` is defined:

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

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

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

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

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

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

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

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

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

38 Otherwise:

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

41 **19.1.1.3 Returns**

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

44 **19.1.1.4 Errors**

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

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

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

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

50 **19.1.1.5 Cross-References**

51 `posix_madvise()`, 19.2.1.

52 **19.1.2 File Space Control**

53 Function: `posix_fallocate()`.

54 **19.1.2.1 Synopsis**

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

56 #include `<fcntl.h>`

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

58 **19.1.2.2 Description**

59 If `{_POSIX_ADVISORY_INFO}` is defined:

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

65 If the *offset + len* is beyond the current file size, then *posix_fallocate()* shall
 66 adjust the file size to *offset + len*. Otherwise, the file size shall not be
 67 changed.

68 It is implementation defined whether a previous *posix_fadvise()* call
 69 influences allocation strategy.

70 Space allocated via *posix_fallocate()* shall be freed by a successful call to
 71 *creat()* or *open()* that truncates the size of the file. Space allocated via
 72 *posix_fallocate()* may be freed by a successful call to *ftruncate()* that
 73 reduces the file size to a size smaller than *offset + len*.

74 Otherwise:

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

77 **19.1.2.3 Returns**

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

80 **19.1.2.4 Errors**

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

- | | | |
|----|----------|--|
| 83 | [EBADF] | The <i>fd</i> argument is not a valid file descriptor. |
| 84 | [EBADF] | The <i>fd</i> argument references a file that was opened without write |
| 85 | | permission. |
| 86 | [EFBIG] | The value of <i>offset + len</i> is greater than the maximum file size. |
| 87 | [EINTR] | A signal was caught during execution. |
| 88 | [EINVAL] | The <i>len</i> argument was zero or the <i>offset</i> argument was less than |
| 89 | | zero. |
| 90 | [EIO] | An I/O error occurred while reading from or writing to a file |
| 91 | | system. |
| 92 | [ENODEV] | The <i>fd</i> argument does not refer to a regular file. |
| 93 | [ENOSPC] | There is insufficient free space remaining on the file system |
| 94 | | storage media. |
| 95 | [ESPIPE] | The <i>fd</i> argument is associated with a pipe or FIFO. |

96 **19.1.2.5 Cross-References**

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

98 **19.2 Memory Advisory Information and Alignment Control**

99 **19.2.1 Memory Advisory Information**

100 Function: *posix_madvise()*.

101 **19.2.1.1 Synopsis**

```
102 #include <sys/types.h>
103 #include <sys/mman.h>
104 int posix_madvise(void *addr, size_t len, int advice);
```

105 **19.2.1.2 Description**

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

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

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

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

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

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

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

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

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

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

136 Otherwise:

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

139 **19.2.1.3 Returns**

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

142 **19.2.1.4 Errors**

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

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

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

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

151 [EINVAL] The value of *addr* is not a multiple of the value returned by *sys-*
 152 *conf()* when the *name* value `_SC_PAGESIZE` is used.

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

154 **19.2.1.5 Cross-References**

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

156 **19.2.2 Aligned Memory Allocation**

157 Function: *posix_memalign()*.

158 **19.2.2.1 Synopsis**

```
159 #include <sys/types.h>
160 #include <stdlib.h>
161 int posix_memalign(void **memptr, size_t alignment,
162                  size_t size);
```

163 **19.2.2.2 Description**

164 If `{_POSIX_ADVISORY_INFO}` is defined:

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

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

172 Otherwise:

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

175 **19.2.2.3 Returns**

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

179 **19.2.2.4 Errors**

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

182 [EINVAL] The value of the *alignment* parameter is not a power of two mul-
 183 tiple of *sizeof(void *)*.

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

186 **19.2.2.5 Cross-References**

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

Annex A (informative) Bibliography

1 **A.2 Other Standards**

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

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

6 **A.3 Historical Documentation and Introductory Texts**

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

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

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 items in B.2.3:*

6 **B.2.3.1 measurement of execution time**

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
 19 close, dup2, and open operations to be used by *posix_spawn()* or *posix_spawnp()*
 20 to arrive at the set of open file descriptors inherited by the spawned process from the
 21 set of open file descriptors in the parent at the time of the *posix_spawn()* or
 22 *posix_spawnp()* call. It had been suggested that the close and dup2 operations
 23 alone are sufficient to rearrange file descriptors and that files which need be
 24 opened for use by the spawned process can be handled either by having the calling

25 process open them before the *posix_spawn()* or *posix_spawnnp()* call (and close
 26 them after) or by passing file names to the spawned process (in *argv*) so that it
 27 may open them itself. The working group recommends that applications use one of
 28 these two methods when practical since detailed error status on a failed open
 29 operation is always available to the application this way. However, the working
 30 group feels that allowing a spawn file actions object to specify open operations is
 31 still appropriate because

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

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

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

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

56 ***File Descriptor Mapping Rationale***

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

65 It was noted by a member of the Ada Language Bindings working group that the
 66 approved Ada Language *Start_Process* family of POSIX process primitives uses a
 67 caller-specified set of file actions to alter the normal *fork()* / *exec* semantics for
 68 inheritance of file descriptors in a very flexible way, yet no such problems exist
 69 because the burden of determining how to achieve the final file descriptor map-

70 ping is completely on the application. Furthermore, although the file actions inter-
71 face appears frightening at first glance, it is actually quite simple to implement in
72 either a library or the kernel.

73 **B.3.1.5 Spawn Attributes**

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

86 **B.3.1.6 Spawn a Process**

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

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

96 This function shall be called `posix_spawn()`. A closely related function,
97 `posix_spawnnp()`, is included for completeness.

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

103 This view of the role of `posix_spawn()` and `posix_spawnnp()` influenced the design of
104 their API. It does not attempt to provide the full functionality of `fork()` / `exec` in
105 which arbitrary user-specified operations of any sort are permitted between the
106 creation of the child process and the execution of the new process image; any
107 attempt to reach that level would need to provide a programming language as
108 parameters. Instead, `posix_spawn()` and `posix_spawnnp()` are process creation
109 primitives like the `Start_Process` and `Start_Process_Search` Ada language

110 bindings in ISO/IEC 14519:1998 {B1} package `POSIX_Process_Primitives` and
111 also like those in many operating systems that are not UNIX¹⁾ systems, but with
112 some POSIX-specific additions.

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

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

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

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

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

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

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

152

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

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

158 **Requirements**

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

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

162 Additional goals are the following:

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

169 **Two-Syntax Rationale**

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

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

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

179 The `Start_Process` and `Start_Process_Search` procedures from ISO/IEC
 180 14519:1998 {B1}, the Ada Language Binding to POSIX.1, encapsulate *fork()* and
 181 *exec* functionality in a manner similar to that of *posix_spawn()* and
 182 *posix_spawnp()*. Originally, in keeping with its simplicity goal, the working group
 183 had limited the capabilities of *posix_spawn()* and *posix_spawnp()* to a subset of
 184 the capabilities of `Start_Process` and `Start_Process_Search`; certain nonde-
 185 fault capabilities were not supported. However, based on suggestions by the ballot
 186 group to improve file descriptor mapping or drop it, and on the advice of an Ada
 187 Bindings working group member, the working group decided that *posix_spawn()*
 188 and *posix_spawnp()* should be sufficiently powerful to implement `Start_Process`
 189 and `Start_Process_Search`. The rationale is that if the Ada language binding to
 190 such a primitive had already been approved as an IEEE standard, there can be lit-
 191 tle justification for not approving the functionally equivalent parts of a C binding.
 192 The only three capabilities provided by *posix_spawn()* and *posix_spawnp()* that
 193 are not provided by `Start_Process` and `Start_Process_Search` are optionally
 194 specifying the child's process group id, the set of signals to be reset to default sig-
 195 nal handling in the child process, and the child's scheduling policy and
 196 parameters.

197 For the Ada Language Binding for `Start_Process` to be implemented with
 198 *posix_spawn()*, that binding would need to explicitly pass an empty signal mask

199 and the parent's environment to *posix_spawn()* whenever the caller of `Start_`
200 `Process` allowed these arguments to default since *posix_spawn()* does not provide
201 such defaults. The ability of `Start_Process` to mask user-specified signals during
202 its execution is functionally unique to the Ada Language Binding and shall be
203 dealt with in the binding separately from the call to *posix_spawn()*.

204 ***Process Group***

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

209 ***Threads***

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

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

228 ***Asynchronous Error Notification Rationale***

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

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

239 The working group had proposed using two additional macros to interpret
240 *stat_val*: First, WIFSPAWNFAIL would have detected a status that indicated that
241 the child exited because of an error detected during the *posix_spawn()* or
242 *posix_spawnp()* operations rather than during actual execution of the child pro-
243 cess image. Second, WSPAWNERRNO would have extracted the error value if
244 WIFSPAWNFAIL indicated a failure. The balloting group strongly opposed this
245 approach because it would make a library implementation of *posix_spawn()* or

246 *posix_spawn()* dependent on kernel modifications to *waitpid()* to be able to
247 embed special information in *stat_val* to indicate a spawn failure.

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

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

270 **Library Implementation of Spawn**

271 The *posix_spawn()* or *posix_spawnnp()* operation is enough to

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

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

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

```

287
288 #include <sys/types.h>
289 #include <stdlib.h>
290 #include <stdio.h>
291 #include <unistd.h>
292 #include <sched.h>
293 #include <fcntl.h>
294 #include <signal.h>
295 #include <errno.h>
296 #include <string.h>
297 #include <signal.h>

298 /*#include <spawn.h>*/
299 /*****
300 /*Things that could be defined in spawn.h*/
301 *****/
302 typedef struct
303 {
304     short posix_attr_flags;
305 #define POSIX_SPAWN_SETPGROUP      0x1
306 #define POSIX_SPAWN_SETSIGMASK    0x2
307 #define POSIX_SPAWN_SETSIGDEF     0x4
308 #define POSIX_SPAWN_SETSCHEDULER  0x8
309 #define POSIX_SPAWN_SETSCHEDPARAM 0x10
310 #define POSIX_SPAWN_RESETPIDS     0x20
311     pid_t  posix_attr_pgroup;
312     sigset_t posix_attr_sigmask;
313     sigset_t posix_attr_sigdefault;
314     int  posix_attr_schedpolicy;
315     struct sched_param posix_attr_schedparam;
316 } posix_spawnattr_t;

317 typedef char *posix_spawn_file_actions_t;

318 int posix_spawn_file_actions_init(
319     posix_spawn_file_actions_t *file_actions);
320 int posix_spawn_file_actions_destroy(
321     posix_spawn_file_actions_t *file_actions);
322 int posix_spawn_file_actions_addclose(
323     posix_spawn_file_actions_t *file_actions,
324     int fildes);
325 int posix_spawn_file_actions_adddup2(
326     posix_spawn_file_actions_t *file_actions,
327     int fildes, int newfildes);
328 int posix_spawn_file_actions_addopen(
329     posix_spawn_file_actions_t *file_actions,
330     int fildes, const char *path, int oflag,
331     mode_t mode);
332 int posix_spawnattr_init (
333     posix_spawnattr_t *attr);
334 int posix_spawnattr_destroy (
335     posix_spawnattr_t *attr);
336 int posix_spawnattr_getflags (
337     const posix_spawnattr_t *attr,
338     short *flags);
339 int posix_spawnattr_setflags (

```

```

340         posix_spawnattr_t *attr,
341         short flags);
342 int posix_spawnattr_getpgroup (
343     const posix_spawnattr_t *attr,
344     pid_t *pgroup);
345 int posix_spawnattr_setpgroup (
346     posix_spawnattr_t *attr,
347     pid_t pgroup);
348 int posix_spawnattr_getschedpolicy (
349     const posix_spawnattr_t *attr,
350     int *schedpolicy);
351 int posix_spawnattr_setschedpolicy (
352     posix_spawnattr_t *attr,
353     int schedpolicy);
354 int posix_spawnattr_getschedparam (
355     const posix_spawnattr_t *attr,
356     struct sched_param *schedparam);
357 int posix_spawnattr_setschedparam (
358     posix_spawnattr_t *attr,
359     const struct sched_param *schedparam);
360 int posix_spawnattr_getsigmask (
361     const posix_spawnattr_t *attr,
362     sigset_t *sigmask);
363 int posix_spawnattr_setsigmask (
364     posix_spawnattr_t *attr,
365     const sigset_t *sigmask);
366 int posix_spawnattr_getsigdefault (
367     const posix_spawnattr_t *attr,
368     sigset_t *sigdefault);
369 int posix_spawnattr_setsigdefault (
370     posix_spawnattr_t *attr,
371     const sigset_t *sigdefault);
372 int posix_spawn(
373     pid_t *pid,
374     const char *path,
375     const posix_spawn_file_actions_t *file_actions,
376     const posix_spawnattr_t *attrp,
377     char * const argv[],
378     char * const envp[]);
379 int posix_spawn(
380     pid_t *pid,
381     const char *file,
382     const posix_spawn_file_actions_t *file_actions,
383     const posix_spawnattr_t *attrp,
384     char * const argv[],
385     char * const envp[]);

386 /*****/
387 /*Example posix_spawn() library routine*/
388 /*****/
389 int posix_spawn(pid_t *pid,
390     const char *path,
391     const posix_spawn_file_actions_t *file_actions,
392     const posix_spawnattr_t *attrp,
393     char * const argv[],
394     char * const envp[])

```

```

395     {
396     /*Create process*/
397     if((*pid=fork()) == (pid_t)0)
398     {
399         /*This is the child process*/
400         /*Worry about process group*/
401         if(attrp->posix_attr_flags & POSIX_SPAWN_SETPGROUP)
402         {
403             /*Override inherited process group*/
404             if(setpgid(0, attrp->posix_attr_pgroup) != 0)
405             {
406                 /*Failed*/
407                 exit(127);
408             }
409         }

410         /*Worry about process signal mask*/
411         if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGMASK)
412         {
413             /*Set the signal mask (can't fail)*/
414             sigprocmask(SIG_SETMASK, &attrp->posix_attr_sigmask,
415                 NULL);
416         }

417         /*Worry about resetting effective user and group IDs*/
418         if(attrp->posix_attr_flags & POSIX_SPAWN_RESETIDS)
419         {
420             /*None of these can fail for this case.*/
421             setuid(getuid());
422             setgid(getgid());
423         }

424         /*Worry about defaulted signals*/
425         if(attrp->posix_attr_flags & POSIX_SPAWN_SETSIGDEF)
426         {
427             struct sigaction deflt;
428             sigset_t all_signals;
429             int s;

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

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

435             /*Loop for all signals*/
436             for(s=0; sigismember(&all_signals,s); s++)
437             {
438                 /*Signal to be defaulted?*/
439                 if(sigismember(&attrp->posix_attr_sigdefault,s))
440                 {
441                     /*Yes - default this signal*/
442                     if(sigaction(s, &deflt, NULL) == -1)
443                     {
444                         /*Failed*/

```

```

445         exit(127);
446     }
447 }
448 }
449 }

450 /*Worry about the fds if we are to map them*/
451 if(file_actions != NULL)
452 {
453     /*Loop for all actions in object *file_actions*/
454     /*(implementation dives beneath abstraction)*/
455     char *p = *file_actions;
456     while(*p != '\0')
457     {
458         if(strncmp(p,"close(",6) == 0)
459         {
460             int fd;
461             if(sscanf(p+6,"%d",&fd) != 1)
462             {
463                 exit(127);
464             }
465             if(close(fd) == -1) exit(127);
466         }
467         else if(strncmp(p,"dup2(",5) == 0)
468         {
469             int fd,newfd;
470             if(sscanf(p+5,"%d,%d",&fd,&newfd) != 2)
471             {
472                 exit(127);
473             }
474             if(dup2(fd, newfd) == -1) exit(127);
475         }
476         else if(strncmp(p,"open(",5) == 0)
477         {
478             int fd,oflag;
479             mode_t mode;
480             int tempfd;
481             char path[1000]; /*should be dynamic*/
482             char *q;
483             if(sscanf(p+5,"%d",&fd) != 1)
484             {
485                 exit(127);
486             }
487             p = strchr(p, ',') + 1;
488             q = strchr(p, '*');
489             if(q == NULL) exit(127);
490             strncpy(path, p, q-p);
491             path[q-p] = '\0';
492             if(sscanf(q+1,"%o,%o",&oflag,&mode)!=2)
493             {
494                 exit(127);
495             }
496             if(close(fd) == -1)
497             {
498                 if(errno != EBADF) exit(127);
499             }

```

```

500         tempfd = open(path, oflag, mode);
501         if(tempfd == -1) exit(127);
502         if(tempfd != fd)
503             {
504                 if(dup2(tempfd,fd) == -1)
505                     {
506                         exit(127);
507                     }
508                 if(close(tempfd) == -1)
509                     {
510                         exit(127);
511                     }
512             }
513         }
514     else
515     {
516         exit(127);
517     }
518     p = strchr(p, ' ') + 1;
519 }
520 }

521     /*Worry about setting new scheduling policy and parameters*/
522     if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDULER)
523     {
524         if(sched_setscheduler(0, attrp->posix_attr_schedpolicy,
525             &attrp->posix_attr_schedparam) == -1)
526             {
527                 exit(127);
528             }
529     }

530     /*Worry about setting only new scheduling parameters*/
531     if(attrp->posix_attr_flags & POSIX_SPAWN_SETSCHEDPARAM)
532     {
533         if(sched_setparam(0, &attrp->posix_attr_schedparam)==-1)
534             {
535                 exit(127);
536             }
537     }

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

550     /******
551     /* Here is a crude but effective implementation of the */

```

```

552 /* file action object operators which store actions as */
553 /* concatenated token separated strings. */
554 /******
555 /*Create object with no actions.*/
556 int posix_spawn_file_actions_init(
557     posix_spawn_file_actions_t *file_actions)
558     {
559     *file_actions = malloc(sizeof(char));
560     if(*file_actions == NULL) return ENOMEM;
561     strcpy(*file_actions, "");
562     return 0;
563     }

564 /*Free object storage and make invalid.*/
565 int posix_spawn_file_actions_destroy(
566     posix_spawn_file_actions_t *file_actions)
567     {
568     free(*file_actions);
569     *file_actions = NULL;
570     return 0;
571     }

572 /*Add a new action string to object.*/
573 static int add_to_file_actions(
574     posix_spawn_file_actions_t *file_actions,
575     char *new_action)
576     {
577     *file_actions = realloc
578     (*file_actions, strlen(*file_actions)+strlen(new_action)+1);
579     if(*file_actions == NULL) return ENOMEM;
580     strcat(*file_actions, new_action);
581     return 0;
582     }

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

592 /*Add a dup2 action to object.*/
593 int posix_spawn_file_actions_adddup2(
594     posix_spawn_file_actions_t *file_actions,
595     int fildes, int newfildes)
596     {
597     char temp[100];
598     sprintf(temp, "dup2(%d,%d)", fildes, newfildes);
599     return add_to_file_actions(file_actions, temp);
600     }

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

```

```

603         posix_spawn_file_actions_t *file_actions,
604         int fildes, const char *path, int oflag,
605         mode_t mode)
606     {
607         char temp[100];
608         sprintf(temp, "open(%d,%s*%o,%o)", fildes, path, oflag, mode);
609         return add_to_file_actions(file_actions, temp);
610     }

611     /******
612     /* Here is a crude but effective implementation of the */
613     /* spawn attributes object functions which manipulate */
614     /* the individual attributes. */
615     /******
616     /*Initialize object with default values.*/
617     int posix_spawnattr_init (
618         posix_spawnattr_t *attr)
619     {
620         attr->posix_attr_flags=0;
621         attr->posix_attr_pgroup=0;
622         /* Default value of signal mask is the parent's signal mask */
623         /* other values are also allowed */
624         sigprocmask(0,NULL,&attr->posix_attr_sigmask);
625         sigemptyset(&attr->posix_attr_sigdefault);
626         /* Default values of scheduling attr. inherited from the parent */
627         /* other values are also allowed */
628         attr->posix_attr_schedpolicy=sched_getscheduler(0);
629         sched_getparam(0,&attr->posix_attr_schedparam);
630         return 0;
631     }

632     int posix_spawnattr_destroy (
633         posix_spawnattr_t *attr)
634     {
635         /* No action needed */
636         return 0;
637     }

638     int posix_spawnattr_getflags (
639         const posix_spawnattr_t *attr,
640         short *flags)
641     {
642         *flags=attr->posix_attr_flags;
643         return 0;
644     }

645     int posix_spawnattr_setflags (
646         posix_spawnattr_t *attr,
647         short flags)
648     {
649         attr->posix_attr_flags=flags;
650         return 0;
651     }

652     int posix_spawnattr_getpgroup (
653         const posix_spawnattr_t *attr,

```



```
654         pid_t *pgroup)
655     {
656     *pgroup=attr->posix_attr_pgroup;
657     return 0;
658     }

659 int posix_spawnattr_setpgroup (
660     posix_spawnattr_t *attr,
661     pid_t pgroup)
662     {
663     attr->posix_attr_pgroup=pgroup;
664     return 0;
665     }

666 int posix_spawnattr_getschedpolicy (
667     const posix_spawnattr_t *attr,
668     int *schedpolicy)
669     {
670     *schedpolicy=attr->posix_attr_schedpolicy;
671     return 0;
672     }

673 int posix_spawnattr_setschedpolicy (
674     posix_spawnattr_t *attr,
675     int schedpolicy)
676     {
677     attr->posix_attr_schedpolicy=schedpolicy;
678     return 0;
679     }

680 int posix_spawnattr_getschedparam (
681     const posix_spawnattr_t *attr,
682     struct sched_param *schedparam)
683     {
684     *schedparam=attr->posix_attr_schedparam;
685     return 0;
686     }

687 int posix_spawnattr_setschedparam (
688     posix_spawnattr_t *attr,
689     const struct sched_param *schedparam)
690     {
691     attr->posix_attr_schedparam=*schedparam;
692     return 0;
693     }

694 int posix_spawnattr_getsigmask (
695     const posix_spawnattr_t *attr,
696     sigset_t *sigmask)
697     {
698     *sigmask=attr->posix_attr_sigmask;
699     return 0;
700     }

701 int posix_spawnattr_setsigmask (
702     posix_spawnattr_t *attr,
```

```

703         const sigset_t *sigmask)
704     {
705     attr->posix_attr_sigmask=*sigmask;
706     return 0;
707     }

708 int posix_spawnattr_getsigdefault (
709     const posix_spawnattr_t *attr,
710     sigset_t *sigdefault)
711     {
712     *sigdefault=attr->posix_attr_sigdefault;
713     return 0;
714     }

715 int posix_spawnattr_setsigdefault (
716     posix_spawnattr_t *attr,
717     const sigset_t *sigdefault)
718     {
719     attr->posix_attr_sigdefault=*sigdefault;
720     return 0;
721     }
722

```

723 **Figure B-1 – *posix_spawn()* Equivalent**

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

```

727
728 /* To redirect new standard output (fd 1) to a file, */
729 /* and redirect new standard input (fd 0) from my fd socket_pair[1], */
730 /* and close my fd socket_pair[0] in the new process. */
731 posix_spawn_file_actions_t file_actions;
732 posix_spawn_file_actions_init (&file_actions);
733 posix_spawn_file_actions_addopen (&file_actions, 1, "newout", ...);
734 posix_spawn_file_actions_dup2 (&file_actions, socket_pair[1], 0);
735 posix_spawn_file_actions_close (&file_actions, socket_pair[0]);
736 posix_spawn_file_actions_close (&file_actions, socket_pair[1]);
737 posix_spawn(..., &file_actions, ...)
738 posix_spawn_file_actions_destroy (&file_actions);
739

```

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

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

```
742 _____  
743 Save = getuid();  
744 setuid(newid);  
745 posix_spawn(...)  
746 setuid(Save);  
747 _____
```

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

749 **B.13 Execution Scheduling**

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

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

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

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

780 The interface that appears in this section defines a new scheduling policy for
781 threads and processes that behaves according to the rules of the sporadic server
782 mechanism. Scheduling attributes are defined and functions are provided to allow

783 the user to set and get the parameters that control the scheduling behavior of this
784 mechanism, namely the normal and low priority, the replenishment period, the
785 maximum number of pending replenishment operations, and the initial
786 execution-time budget.

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

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

- 792 — The effects of an aperiodic activity on the response time of lower priority
793 activities must be controllable and predictable.
- 794 — The system must provide the fastest possible response time to aperiodic
795 events.
- 796 — It must be possible to take advantage of all the available processing
797 bandwidth not needed by time-critical activities to enhance average-case
798 response times to aperiodic events.

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

- 801 — Background processing consists of assigning a very low priority to the pro-
802 cessing of aperiodic events. It utilizes all the available bandwidth in the sys-
803 tem that has not been consumed by higher priority threads. However, it is
804 difficult, or impossible, to meet requirements on average-case response time
805 because the aperiodic entity has to wait for the execution of all other enti-
806 ties that have higher priority.
- 807 — Polling consists of creating a periodic process or thread for servicing
808 aperiodic requests. At regular intervals, the polling entity is started, and it
809 services accumulated pending aperiodic requests. If no aperiodic requests
810 are pending, the polling entity suspends itself until its next period. Polling
811 allows the aperiodic requests to be processed at a higher priority level.
812 However, worst and average-case response times of polling entities are a
813 direct function of the polling period, and there is execution overhead for
814 each polling period, even if no event has arrived. If the deadline of the
815 aperiodic activity is short compared to the interarrival time, the polling fre-
816 quency must be increased to guarantee meeting the deadline. For this case,
817 the increase in frequency can dramatically reduce the efficiency of the sys-
818 tem and, therefore, its capacity to meet all deadlines. Yet, polling
819 represents a good way to handle a large class of practical problems because
820 it preserves system predictability and because the amortised overhead
821 drops as load increases.
- 822 — Direct event execution consists of executing the aperiodic events at a high
823 fixed-priority level. Typically, the aperiodic event is processed by an inter-
824 rupt service routine as soon as it arrives. This technique provides predict-
825 able response times for aperiodic events, but makes the response times of
826 all lower priority activities completely unpredictable under burst arrival
827 conditions. Therefore, if the density of aperiodic event arrivals is
828 unbounded, it may be a dangerous technique for time-critical systems. Yet,

829 for cases in which the physics of the system imposes a bound on the event
830 arrival rate, it is probably the most efficient technique.

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

841 **B.13.3.2 Existing Practice**

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

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

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

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

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

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

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

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

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

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

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

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

915 **B.14 Clocks and Timers**

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

917 **B.14.3 Execution Time Monitoring**

918 **B.14.3.1 Introduction**

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

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

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

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

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

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

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

982 **B.14.3.3 Overhead Considerations**

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

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

995 The mechanism used to measure the execution time of processes and threads is
996 specified in this document as implementation defined. The reason for this
997 requirement is that both the underlying hardware and the implementation archi-
998 tecture have a very strong influence on the accuracy achievable for measuring
999 CPU-time. For some implementations, the specification of strict accuracy require-
1000 ments would represent very large overheads or even the impossibility of being
1001 implemented.

1002 Since the mechanism for measuring execution time is implementation defined,
1003 realtime applications will be able to take advantage of accurate implementations
1004 using a portable interface. Of course, strictly conforming applications cannot rely
1005 on any particular degree of accuracy, in the same way as they cannot rely on a
1006 very accurate measurement of wall clock time. There will always exist applica-
1007 tions whose accuracy or efficiency requirements on the implementation are more
1008 rigid than the values defined in this or any other standard.

1009 In any case, realtime applications would expect a minimum set of characteristics
1010 from most implementations. One such characteristic is that the sum of all the
1011 execution times of all the threads in a process equals the process execution time
1012 when no CPU-time clocks are disabled. This property may not always be true
1013 because implementations may differ in how they account for time during context
1014 switches. Another characteristic is that the sum of the execution times of all
1015 processes in a system equals the number of processors, multiplied by the elapsed
1016 time, assuming that no processor is idle during that elapsed time. However, in
1017 some systems it might not be possible to relate CPU-time to elapsed time. For
1018 example, in a heterogeneous multiprocessor system in which each processor runs
1019 at a different speed, an implementation may choose to define each “second” of
1020 CPU-time to be a certain number of “cycles” that a CPU has executed.

1021 **B.14.3.5 Existing Practice**

1022 Measuring and limiting the execution time of each concurrent activity are com-
1023 mon features of most industrial implementations of realtime systems. Almost all
1024 critical realtime systems are currently built upon a cyclic executive. With this
1025 approach, a regular timer interrupt kicks off the next sequence of computations.
1026 It also checks that the current sequence has completed. If it has not, then some
1027 error recovery action can be undertaken (or at least an overrun is avoided).
1028 Current software engineering principles and the increasing complexity of software
1029 are driving application developers to implement these systems on multi-threaded
1030 or multi-process operating systems. Therefore, if a POSIX operating system is to be
1031 used for this type of application, then it must offer the same level of protection.

1032 Execution time clocks are also common in most UNIX implementations, although
1033 these clocks usually have different requirements from those of realtime applica-
1034 tions. The POSIX.1 *times()* function supports the measurement of the execution
1035 time of the calling process and its terminated child processes. This execution time
1036 is measured in clock ticks and is supplied as two different values with the user
1037 and system execution times, respectively. BSD {B60} supports the function
1038 *getrusage()*, which allows the calling process to get information about the
1039 resources used by itself and/or all of its terminated child processes. The resource
1040 usage includes user and system CPU time. Some UNIX systems have options to
1041 specify high resolution (up to one microsecond) CPU time clocks using the *times()*
1042 or the *getrusage()* functions.

1043 The *times()* and *getrusage()* interfaces do not meet important realtime require-
1044 ments such as the possibility of monitoring execution time from a different process
1045 or thread or the possibility of detecting an execution time overrun. The latter
1046 requirement is supported in some UNIX implementations that are able to send a
1047 signal when the execution time of a process has exceeded some specified value. For
1048 example, BSD defines the functions *getitimer()* and *setitimer()*, which can operate
1049 either on a realtime clock (wall clock) or on virtual-time or profile-time clocks,

1050 which measure CPU time in two different ways. These functions do not support
1051 access to the execution time of other processes. System V supports similar func-
1052 tions after release 4. Some emerging implementations of threads also support
1053 these functions.

1054 IBM's MVS operating system supports per-process and per-thread execution time
1055 clocks. It also supports limiting the execution time of a given process.

1056 Given all this existing practice, the working group considered that the POSIX.1b
1057 clocks and timers interface was appropriate to meet most of the requirements that
1058 realtime applications have for execution time clocks. Functions were added to get
1059 the CPU time clock IDs and to allow or disallow the thread CPU time clocks (in
1060 order to preserve the efficiency of some implementations of threads).

1061 **B.14.3.6 Clock Constants**

1062 The definition of the manifest constants `CLOCK_PROCESS_CPUTIME_ID` and
1063 `CLOCK_THREAD_CPUTIME_ID` allows processes or threads, respectively, to access
1064 their own execution-time clocks. However, given a process or thread, access to its
1065 own execution-time clock is also possible if the clock ID of this clock is obtained
1066 through a call to `clock_getcpuclockid()` or `pthread_getcpuclockid()`. Therefore,
1067 these constants are not necessary and could be deleted to make the interface
1068 simpler. Their existence saves one system call in the first access to the CPU-time
1069 clock of each process or thread. The working group considered this issue and
1070 decided to leave the constants in the standard because they are closer to the
1071 POSIX.1b use of clock identifiers.

1072 **B.14.3.7 Library Implementations of Threads**

1073 In library implementations of threads, kernel entities and library threads can
1074 coexist. In this case, if the CPU-time clocks are supported, most of the clock and
1075 timer functions will need to have two implementations: one in the thread library
1076 and one in the system calls library. The main difference between these two imple-
1077 mentations is that the thread library implementation will have to deal with clocks
1078 and timers that reside in the thread space, while the kernel implementation will
1079 operate on timers and clocks that reside in kernel space. In the library implemen-
1080 tation, if the clock ID refers to a clock that resides in the kernel, a kernel call will
1081 have to be made. The correct version of the function can be chosen by specifying
1082 the appropriate order for the libraries during the link process.

1083 **B.14.3.8 History of Resolution Issues: Deletion of the `enable` attribute**

1084 In the draft corresponding to the first balloting round, CPU-time clocks had an
1085 attribute called `enable`. This attribute was introduced by the working group to
1086 allow implementations to avoid the overhead of measuring execution time for
1087 processes or threads for which this measurement was not required. However, the
1088 `enable` attribute received several ballot objections. The main objection was that
1089 processes are already required to measure execution time by the POSIX.1 `times()`
1090 function. Consequently, the `enable` attribute was considered unnecessary and
1091 was deleted from this standard.

1092 **B.14.4 Rationale Relating to Timeouts**

1093 **B.14.4.1 Requirements for Timeouts**

1094 Realtime systems that have to operate reliably over extended periods without
1095 human intervention are characteristic in embedded applications such as avionics,
1096 machine control, and space exploration, as well as more mundane applications
1097 such as cable TV, security systems, and plant automation. A multi-tasking para-
1098 digm, in which many independent and/or cooperating software functions relin-
1099 quish the processor(s) while waiting for a specific stimulus, resource, condition, or
1100 operation completion, is very useful in producing well-engineered programs for
1101 such systems. For such systems to be robust and fault tolerant, expected
1102 occurrences that are unduly delayed or that never occur must be detected so that
1103 appropriate recovery actions may be taken. This requirement is difficult to
1104 achieve if there is no way for a task to regain control of a processor once it has
1105 relinquished control (blocked) awaiting an occurrence which, perhaps because of
1106 corrupted code, hardware malfunction, or latent software bugs, will not happen
1107 when expected. Therefore, the common practice in realtime operating systems is
1108 to provide a capability to time out such blocking services. Although there are
1109 several methods already defined by POSIX to achieve this timeout capability, none
1110 is as reliable or efficient as initiating a timeout simultaneously with initiating a
1111 blocking service. Timeouts are especially critical in hard-realtime embedded sys-
1112 tems because the processors typically have little time reserve, and allowed fault
1113 recovery times are measured in milliseconds rather than seconds.

1114 The working group largely agreed that such timeouts were necessary and ought to
1115 become part of the standard, particularly vendors of realtime operating systems
1116 whose customers had already expressed a strong need for timeouts. There was
1117 some resistance to inclusion of timeouts in the standard because the desired
1118 effect, fault tolerance, could, in theory, be achieved using existing facilities and
1119 alternative software designs, but there was no compelling evidence that realtime
1120 system designers would embrace such designs at the sacrifice of performance
1121 and/or simplicity.

1122 **B.14.4.2 Which Services Should Be Timed Out?**

1123 Originally, the working group considered the prospect of providing timeouts on *all*
1124 blocking services, including those currently existing in POSIX.1, POSIX.1b, and
1125 POSIX.1c, and future interfaces to be defined by other working groups, as a gen-
1126 eral policy. This proposal was rather quickly rejected because of the scope of such
1127 a change, and the fact that many of those services would not normally be used in a
1128 realtime context. More traditional time-sharing solutions to time out would
1129 suffice for most of the POSIX.1 interfaces, while others had asynchronous alterna-
1130 tives that, while more complex to utilize, would be adequate for some realtime and
1131 all nonrealtime applications.

1132 The list of potential candidates for timeouts was narrowed to the following for
1133 further consideration:

1134 POSIX.1b

1135 — *sem_wait()*1136 — *mq_receive()*1137 — *mq_send()*1138 — *lio_listio()*1139 — *aio_suspend()*1140 — *sigwait()*1141 timeout already implemented by *sigtimedwait()*

1142 POSIX.1c

1143 — *pthread_mutex_lock()*1144 — *pthread_join()*1145 — *pthread_cond_wait()*1146 timeout already implemented by *pthread_cond_timedwait()*

1147 POSIX.1

1148 — *read()*1149 — *write()*

1150 After further review by the working group, the *read()*, *write()*, and *lio_listio()*
 1151 functions (all forms of blocking synchronous I/O) were eliminated from the list
 1152 because

1153 (1) Asynchronous alternatives exist,

1154 (2) Timeouts can be implemented, albeit nonportably, in device drivers, and

1155 (3) A strong desire existed not to introduce modifications to POSIX.1 inter-
 1156 faces.

1157 The working group ultimately rejected *pthread_join()* since both that interface
 1158 and a timed variant of that interface are nonminimal and may be implemented as
 1159 a library function. See B.14.4.3 for a library implementation of *pthread_join()*.

1160 Thus there was a consensus among the working group members to add timeouts
 1161 to 4 of the remaining 5 functions (the timeout for *aio_suspend()* was ultimately
 1162 added directly to POSIX.1b, while the others are added here in POSIX.1d). How-
 1163 ever, *pthread_mutex_lock()* remained contentious.

1164 Many balloting group members feel that *pthread_mutex_lock()* falls into the same
 1165 class as the other functions; that is, it is desirable to time out a mutex lock
 1166 because a mutex may fail to be unlocked due to errant or corrupted code in a criti-
 1167 cal section (looping or branching outside of the unlock code) and, therefore, is
 1168 equally in need of a reliable, simple, and efficient timeout. In fact, since mutexes
 1169 are intended to guard small critical sections, most *pthread_mutex_lock()* calls
 1170 would be expected to obtain the lock without blocking nor utilizing any kernel ser-
 1171 vice, even in implementations of threads with global contention scope; the timeout
 1172 alternative need only be considered after it is determined that the thread shall
 1173 block.

1174 Those opposed to timing out mutexes feel that the very simplicity of the mutex is
 1175 compromised by adding a timeout semantic and that to do so is senseless. They
 1176 claim that if a timed mutex is really deemed useful by a particular application,
 1177 then it can be constructed from the facilities already in POSIX.1b and POSIX.1c.
 1178 The two C language library implementations of mutex locking with timeout in Fig-
 1179 ure B-4 and Figure B-5 represent the solutions offered (in both implementations,
 1180 the timeout parameter is specified as absolute time, not relative time as in the
 1181 proposed POSIX.1c interfaces):

```

1182
1183 #include <pthread.h>
1184 #include <time.h>
1185 #include <errno.h>
1186
1187 int pthread_mutex_timedlock(pthread_mutex_t *mutex,
1188                             const struct timespec *timeout)
1189 {
1190     struct timespec timenow;
1191
1192     while (pthread_mutex_trylock(mutex) == EBUSY)
1193     {
1194         clock_gettime(CLOCK_REALTIME, &timenow);
1195         if (timespec_cmp(&timenow, timeout) >= 0)
1196         {
1197             return ETIMEDOUT;
1198         }
1199         pthread_yield();
1200     }
1201     return 0;
1202 }

```

1202 **Figure B-4 – Spinlock Implementation**

1203 The spinlock implementation is generally unsuitable for any application using
 1204 priority based thread scheduling policies such as {SCHED_FIFO} or {SCHED_RR}.
 1205 The reason is that the mutex could currently be held by a thread of lower priority
 1206 within the same allocation domain; but, since the waiting thread never blocks,
 1207 only threads of equal or higher priority will ever run. Therefore, the mutex can-
 1208 not be unlocked. Setting priority inheritance or priority ceiling protocol on the
 1209 mutex does not solve this problem, since the priority of a mutex-owning thread is
 1210 only boosted if higher priority threads are blocked waiting for the mutex, clearly
 1211 not the case for this spinlock.

1212 The condition wait implementation effectively substitutes the
 1213 *pthread_cond_timedwait()* function (which is currently timed out) for the desired
 1214 *pthread_mutex_timedlock()*. Since waits on condition variables currently do not
 1215 include protocols that avoid priority inversion, this method is generally unsuitable
 1216 for realtime applications because it does not provide the same priority inversion
 1217 protection as the untimed *pthread_mutex_lock()*. Also, for any given implementa-
 1218 tions of the current mutex and condition variable primitives, this library imple-
 1219 mentation has a performance cost at least 2.5 times that of the untimed
 1220 *pthread_mutex_lock()* even in the case where the timed mutex is readily locked
 1221 without blocking. Even in uniprocessors or where assignment is atomic, at least
 1222 an additional *pthread_cond_signal()* is required. In this case,

```

1223
1224 #include <pthread.h>
1225 #include <time.h>
1226 #include <errno.h>

1227 struct timed_mutex
1228     {
1229     int locked;
1230     pthread_mutex_t mutex;
1231     pthread_cond_t cond;
1232     };
1233 typedef struct timed_mutex timed_mutex_t;

1234 int timed_mutex_lock(timed_mutex_t *tm,
1235                    const struct timespec *timeout)
1236     {
1237     int timedout=FALSE;
1238     int error_status;

1239     pthread_mutex_lock(&tm->mutex);

1240     while (tm->locked && !timedout)
1241     {
1242         if ((error_status=pthread_cond_timedwait(&tm->cond,
1243                                                &tm->mutex,
1244                                                timeout))!=0)
1245         {
1246             if (error_status==ETIMEDOUT) timedout = TRUE;
1247         }
1248     }

1249     if(timedout)
1250     {
1251         pthread_mutex_unlock(&tm->mutex);
1252         return ETIMEDOUT;
1253     }
1254     else
1255     {
1256         tm->locked = TRUE;
1257         pthread_mutex_unlock(&tm->mutex);
1258         return 0;
1259     }
1260 }

1261 void timed_mutex_unlock(timed_mutex_t *tm)
1262     {
1263     pthread_mutex_lock(&tm->mutex); /*for case assignment not atomic*/
1264     tm->locked = FALSE;
1265     pthread_mutex_unlock(&tm->mutex);
1266     pthread_cond_signal(&tm->cond);
1267     }
1268

```

Figure B-5 – Condition Wait Implementation

1270 *pthread_mutex_timedlock()* could be implemented at effectively no performance
1271 penalty because the timeout parameters need only be considered after it is deter-
1272 mined that the mutex cannot be locked immediately.

1273 Thus it has not yet been shown that the full semantics of mutex locking with
 1274 timeout can be efficiently and reliably achieved using existing interfaces. Even if
 1275 the existence of an acceptable library implementation were proven, it is difficult to
 1276 justify why the *interface* itself should not be made portable, especially considering
 1277 approval for the other four timeouts.

1278 **B.14.4.3 Rationale for Library Implementation of *pthread_timedjoin()***

1279 The *pthread_join()* C Language example shown in Figure B-6 demonstrates that it
 1280 is possible, using existing *pthread* facilities, to construct a variety of thread that
 1281 allows for joining such a thread, but allows the *join* operation to time out. This
 1282 behavior is achieved by using a *pthread_cond_timedwait()* to wait for the thread
 1283 to exit. A small *timed_thread* descriptor structure is used to pass parameters
 1284 from the creating thread to the created thread and from the exiting thread to the
 1285 joining thread. This implementation is roughly equivalent to what a normal
 1286 *pthread_join()* implementation would do, with the single change being that
 1287 *pthread_cond_timedwait()* is used in place of a simple *pthread_cond_wait()*.

1288 Since it is possible to implement such a facility entirely from existing *pthread*
 1289 interfaces and with roughly equal efficiency and complexity to an implementation
 1290 that would be provided directly by a *threads* implementation, it was the con-
 1291 sensus of the working group members that any *pthread_timedjoin()* facility would
 1292 be unnecessary and should not be provided.

```

1293
1294 /*
1295  * Construct a thread variety entirely from existing functions
1296  * with which a join can be done, allowing the join to time out.
1297  */
1298 #include <pthread.h>
1299 #include <time.h>
1300 struct timed_thread {
1301     pthread_t t;
1302     pthread_mutex_t m;
1303     int exiting;
1304     pthread_cond_t exit_c;
1305     void *(*start_routine)(void *arg);
1306     void *arg;
1307     void *status;
1308 };
1309 typedef struct timed_thread *timed_thread_t;
1310 static pthread_key_t timed_thread_key;
1311 static pthread_once_t timed_thread_once = PTHREAD_ONCE_INIT;
1312 static void timed_thread_init()
1313 {
1314     pthread_key_create(&timed_thread_key, NULL);
1315 }
1316 static void *timed_thread_start_routine(void *args)
1317 /*
1318  * Routine to establish thread specific data value and run the actual
1319  * thread start routine which was supplied to timed_thread_create().
1320  */

```

```

1321 {
1322     timed_thread_t tt = (timed_thread_t) args;

1323     pthread_once(&timed_thread_once, timed_thread_init);
1324     pthread_setspecific(timed_thread_key, (void *)tt);
1325     timed_thread_exit((tt->start_routine)(tt->arg));
1326 }

1327 int timed_thread_create(timed_thread_t ttp, const pthread_attr_t *attr,
1328                        void *(*start_routine)(void *), void *arg)

1329 /*
1330  * Allocate a thread which can be used with timed_thread_join().
1331  */

1332 {
1333     timed_thread_t tt;
1334     int result;

1335     tt = (timed_thread_t) malloc(sizeof(struct timed_thread));
1336     pthread_mutex_init(&tt->m, NULL);
1337     tt->exiting = FALSE;
1338     pthread_cond_init(&tt->exit_c, NULL);
1339     tt->start_routine = start_routine;
1340     tt->arg = arg;
1341     tt->status = NULL;

1342     if ((result = pthread_create(&tt->t, attr,
1343                                timed_thread_start_routine, (void *)tt)) != 0) {
1344         free(tt);
1345         return result;
1346     }

1347     pthread_detach(tt->t);
1348     ttp = tt;
1349     return 0;
1350 }

1351 timed_thread_join(timed_thread_t tt,
1352                  struct timespec *timeout,
1353                  void **status)
1354 {
1355     int result;

1356     pthread_mutex_lock(&tt->m);
1357     result = 0;
1358     /*
1359     * Wait until the thread announces that it's exiting, or until timeout.
1360     */
1361     while (result == 0 && ! tt->exiting) {
1362         result = pthread_cond_timedwait(&tt->exit_c, &tt->m, timeout);
1363     }
1364     pthread_mutex_unlock(&tt->m);
1365     if (result == 0 && tt->exiting) {
1366         *status = tt->status;
1367         free((void *)tt);
1368         return result;
1369     }
1370     return result;
1371 }

```



```

1372 timed_thread_exit(void *status)
1373 {
1374     timed_thread_t tt;
1375     void *specific;

1376     if ((specific=pthread_getspecific(timed_thread_key)) == NULL){
1377         /*
1378          * Handle cases which won't happen with correct usage.
1379          */
1380         pthread_exit(NULL);
1381     }
1382     tt = (timed_thread_t) specific;
1383     pthread_mutex_lock(&tt->m);
1384     /*
1385     * Tell a joiner that we're exiting.
1386     */
1387     tt->status = status;
1388     tt->exiting = TRUE;
1389     pthread_cond_signal(&tt->exit_c);
1390     pthread_mutex_unlock(&tt->m);
1391     /*
1392     * Call pthread exit() to call destructors and really exit the thread.
1393     */
1394     pthread_exit(NULL);
1395 }
1396

```

1397 **Figure B-6** – *pthread_join()* with timeout

1398 **B.14.4.4 Form of the Timeout Interfaces**

1399 The working group considered a number of alternative ways to add timeouts to
1400 blocking services. At first, a system interface that would specify a one-shot or per-
1401 sistent timeout to be applied to subsequent blocking services invoked by the cal-
1402 ling process or thread was considered because it allowed all blocking services to be
1403 timed out in a uniform manner with a single additional interface; this interface
1404 was rather quickly rejected because it could easily result in the wrong services
1405 being timed out.

1406 It was suggested that a timeout value might be specified as an attribute of the
1407 object (e.g., semaphore, mutex, message queue), but there was no consensus on
1408 this suggestion, either on a case-by-case basis or for all timeouts.

1409 Looking at the two existing timeouts for blocking services indicates that the work-
1410 ing group members favor a separate interface for the timed version of a function.
1411 However, *pthread_cond_timedwait()* utilizes an absolute timeout value while
1412 *sigtimedwait()* uses a relative timeout value. The working group members agreed
1413 that relative timeout values are appropriate where the timeout mechanism's pri-
1414 mary use was to deal with an unexpected or error situation, but they are inap-
1415 propriate when the timeout has to expire at a particular time or before a specific
1416 deadline. For the timeouts being introduced in this document, the working group
1417 considered allowing both relative and absolute timeouts as is done with POSIX.1b
1418 timers, but ultimately favored the simpler absolute timeout form.

1419 An absolute time measure can be easily implemented on top of an interface that
1420 specifies relative time by reading the clock, calculating the difference between the

1421 current time and the desired wake up time, and issuing a relative timeout call.
1422 But there is a race condition with this approach because the thread could be
1423 preempted after reading the clock, but before making the timed out call; in this
1424 case, the thread would be awakened later than it should and, thus, if the wake up
1425 time represented a deadline, the thread would miss it.

1426 There is also a race condition when trying to build a relative timeout on top of an
1427 interface that specifies absolute timeouts. In this case, the clock would have to be
1428 read to calculate the absolute wake up time as the sum of the current time plus
1429 the relative timeout interval. In this case, if the thread is preempted after reading
1430 the clock, but before making the timed out call, the thread would be awakened
1431 earlier than desired.

1432 But the race condition with the absolute timeouts interface is not as bad as the
1433 one that happens with the relative timeout interface because there are simple
1434 workarounds. For the absolute timeouts interface, if the timing requirement is a
1435 deadline, it can still be met because the thread woke up earlier than the deadline.
1436 If the timeout is just used as an error recovery mechanism, the precision of timing
1437 is not really important. If the timing requirement is that between actions A and B
1438 a minimum interval of time must elapse, the absolute timeout interface can be
1439 safely used by reading the clock after action A has been started. It could be argued
1440 that, since the call with the absolute timeout is atomic from the application point
1441 of view, it is not possible to read the clock after action A if this action is part of the
1442 timed out call. But for the calls for which timeouts are specified (e.g., locking a
1443 mutex, waiting for a semaphore, waiting for a message, waiting until there is
1444 space in a message queue), the timeouts that an application would build on these
1445 actions would not be triggered by these actions themselves, but by some other
1446 external action. For example, to wait for at least 20 milliseconds for a message to
1447 arrive to a message queue, this time interval would be started by some event that
1448 would trigger both the action that produces the message and the action that waits
1449 for the message to arrive, and not by the wait-for-message operation itself. In this
1450 case, the workaround proposed above could be used.

1451 For these reasons, the absolute timeout is preferred over the relative timeout
1452 interface.

1453 ⇒ **Annex B Rationale and Notes** *Add the following subclause.*

1454 **B.19 Advisory Information**

1455 The POSIX.1b standard contains an informative annex with proposed interfaces
1456 for “realtime files.” These interfaces could determine groups of the exact param-
1457 eters required to do “direct I/O” or “extents.” These interfaces were objected to by
1458 a significant portion of the balloting group as too complex. A portable application
1459 had little chance of correctly navigating the large parameter space to match its
1460 desires to the system. In addition, they only applied to a new type of file (realtime
1461 files) and they told the implementation exactly what to do as opposed to advising
1462 the implementation on application behavior and letting it optimize for the system
1463 on which the (portable) application was running. For example, it was not clear
1464 how a system that had a disk array should set its parameters.

1465 There seemed to be several overall goals:

- 1466 — Optimizing Sequential Access
- 1467 — Optimizing Caching Behavior
- 1468 — Optimizing I/O data transfer
- 1469 — Preallocation

1470 The advisory interfaces, *posix_fadvise()* and *posix_madvise()*, satisfy the first two
 1471 goals. The `POSIX_FADV_SEQUENTIAL` and `POSIX_MADV_SEQUENTIAL` *advice*
 1472 tells the implementation to expect serial access. Typically the system will prefetch
 1473 the next several serial accesses in order to overlap I/O. It may also free previously
 1474 accessed serial data if memory is tight. If the application is not doing serial access,
 1475 it can use `POSIX_FADV_WILLNEED` and `POSIX_MADV_WILLNEED` to accomplish
 1476 I/O overlap, as required. When the application advises `POSIX_FADV_RANDOM` or
 1477 `POSIX_MADV_RANDOM` behavior, the implementation usually tries to fetch a
 1478 minimum amount of data with each request; and it does not expect much locality.
 1479 `POSIX_FADV_DONTNEED` and `POSIX_MADV_DONTNEED` allow the system to free
 1480 up caching resources as the data will not be required in the near future.

1481 `POSIX_FADV_NOREUSE` tells the system that caching the specified data is not
 1482 optimal. For file I/O, the transfer should go directly to the user buffer instead of
 1483 being cached internally by the implementation. To portably perform direct disk
 1484 I/O on all systems, the application shall perform its I/O transfers according to the
 1485 following rules:

- 1486 (1) The user buffer should be aligned according to the `{POSIX_REC_XFER_-`
 1487 `ALIGN}` *pathconf()* variable.
- 1488 (2) The number of bytes transferred in an I/O operation should be a multiple
 1489 of the `{POSIX_ALLOC_SIZE_MIN}` *pathconf()* variable.
- 1490 (3) The offset into the file at the start of an I/O operation should be a multi-
 1491 ple of the `{POSIX_ALLOC_SIZE_MIN}` *pathconf()* variable.
- 1492 (4) The application should ensure that all threads that open a given file
 1493 specify `POSIX_FADV_NOREUSE` to be sure that there is no unexpected
 1494 interaction between threads using buffered I/O and threads using direct
 1495 I/O to the same file.

1496 In some cases, a user buffer should be properly aligned in order to be transferred
 1497 directly to/from the device. The `{POSIX_REC_XFER_ALIGN}` *pathconf()* variable
 1498 tells the application the proper alignment.

1499 The preallocation goal is met by the space control function, *posix_fallocate()*. The
 1500 application can use *posix_fallocate()* to guarantee no `[ENOSPC]` errors and to
 1501 improve performance by prepaying any overhead required for block allocation.

1502 Implementations may use information conveyed by a previous *posix_fadvise()* call
 1503 to influence the manner in which allocation is performed. For example, assume
 1504 an application does the following calls:

```
1505     fd = open("file")
1506     posix_fadvise(fd, offset, len, POSIX_FADV_SEQUENTIAL)
1507     posix_fallocate(fd, len, size)
```

1508 As a result, an implementation might allocate the file contiguously on disk.

1509 Finally, the *pathconf()* variables {POSIX_REC_MIN_XFER_SIZE}, {POSIX_REC_-
1510 MAX_XFER_SIZE} and {POSIX_REC_INCR_XFER_SIZE} tell the application a range
1511 of transfer sizes that are recommended for best I/O performance.

1512 Where bounded response time is required, the vendor can supply the appropriate
1513 settings of the advisories to achieve a guaranteed performance level.

1514 The interfaces meet the goals while allowing applications using regular files to
1515 take advantage of performance optimizations. The interfaces tell the implementa-
1516 tion expected application behavior that the implementation can use to optimize
1517 performance on a particular system with a particular dynamic load.

1518 The *posix_memalign()* function was added to allow for the allocation of specifically
1519 aligned buffers, e.g. for {POSIX_REC_XFER_ALIGN}.

1520 The working group also considered the alternative of adding a function that would
1521 return an aligned pointer to memory within a user supplied buffer. This method
1522 was not considered to be best because it potentially wastes large amounts of
1523 memory when buffers need to be aligned on large alignment boundaries.

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 {19.1.1}	63
<i>posix_fallocate()</i>	File Space Control {19.1.2}	64
<i>posix_madvise()</i>	Memory Advisory Information {19.2.1}	66
<i>posix_memalign()</i>	Aligned Memory Allocation {19.2.2}	67
<i>posix_spawn()</i>	Spawn a Process {3.1.6}	20
<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
<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}.....	20
<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 ... 5
 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 ... 92
 address space ... 67
 Advisory Information ... 63, 102
 Advisory Information option ... 7-8, 10, 27,
 29, 63-64, 66-67
ai_suspend() ... 96
alarm() ... 25
 Aligned Memory Allocation ... 67
 ARG_MAX ... 21
 Asynchronous Input and Output ... 31
 Asynchronous I/O Control Block ... 31
 attributes
 enable ... 94
 schedparam ... 43
 attributes
 cpu-clock-requirement ... 92
 spawn-default ... 18
 spawn-flags ... 18, 20, 22-24
 spawn-pgroup ... 18, 20, 22, 76
 spawn-schedparam ... 19-20, 23
 spawn-schedpolicy ... 19-20, 23
 spawn-sigdefault ... 19-20, 23
 spawn-sigmask ... 18, 20, 23

B

B.2 ... 71
 B-4 ... 97
 B-5 ... 97
 background ... 87-88
 background priority ... 87
 Bibliography ... 69
 blocked thread ... 40

bounded response ... 1, 90, 104
 BSD ... 93

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 ... 94
 clock
 CPU-time ... 5, 13, 47-50, 59, 91-94
clock_getcpuclockid() ... 8, 47, 49-51, 92, 94
 function definition ... 49
clock_getres() ... 50-51
clock_gettime() ... 47, 50-51
 CLOCK_PROCESS_CPUTIME_ID ... 47, 92, 94
 CLOCK_REALTIME ... 33, 36, 54, 56, 97
 clock resolution ... 33, 36, 54, 56
 Clocks ... 47
 CPU-time ... 13
 Clocks and Timers ... 47-48, 91
 clocks
 CPU-Time ... 13
 CPU-time ... 47
clock_settime() ... 41, 47, 50-51
 CLOCK_THREAD_CPUTIME_ID ... 47, 92, 94
close() ... 16, 25
 Compile-Time Symbolic Constants for Portability Specifications ... 10-11
 Condition Wait Implementation ... 98
 Configurable Pathname Variables ... 29
 Configurable System Variables ... 27
 conformance ... 2
 implementation ... 2
 Conformance ... 2
 Conforming Implementation Options ... 2
 CPU ... 5-6, 13, 47-50, 91-94

- cpu-clock-requirement
 - attribute ... 92
 - CPU-time Clock Characteristics ... 48
 - CPU-Time clock ... 13
 - CPU-time clock ... 13, 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
 - C Standard ... 5, 68
 - abbreviation ... 5
 - definition of ... 5
- ## D
- Data Definitions for Asynchronous Input and Output ... 31
 - Definitions ... 5
 - Definitions and General Requirements ... 71
 - document ... 2, 18, 21, 44, 69, 92, 101
 - dup2()* ... 16, 25
 - Dynamically Setting the Sporadic Server Policy ... 90
 - Dynamic Thread Scheduling Parameters
 - Access—Description ... 45
 - Dynamic Thread Scheduling Parameters
 - Access—Errors ... 45
 - Dynamic Thread Scheduling Parameters
 - Access ... 45
- ## E
- [EBADF] ... 16, 64-65
 - EBADF ... 81
 - EBUSY ... 97
 - [EDEADLK] ... 36
 - [EFBIG] ... 65
 - effective group ID ... 23
 - effective user ID ... 23
 - [EINTR] ... 54, 56, 65
 - [EINVAL] ... 16, 20, 24, 34, 36, 54, 56, 64-65, 67-68
 - [EIO] ... 65
 - enable
 - attribute ... 94
 - [ENODEV] ... 65
 - [ENOMEM] ... 16, 20, 67-68
 - ENOMEM ... 83
 - [ENOSPC] ... 65, 103
 - [ENOTSUP] ... 45, 48
 - [EPERM] ... 50
 - [ESPIPE] ... 64-65
 - [ESRCH] ... 50-51
 - [ETIMEDOUT] ... 34, 36, 54, 56
 - ETIMEDOUT ... 97-98
 - exec ... 13, 23-24, 72-75
 - Execute a File—Description ... 13
 - Execute a File ... 13
 - Execution Scheduling ... 39, 87
 - execution time
 - definition of ... 5
 - Execution Time Monitoring ... 48, 91
 - Execution Time Monitoring Interface ... 91
 - Existing Practice ... 89, 93
 - _exit()* ... 25
- ## F
- FALSE ... 98, 100
 - fcntl()* ... 25
 - <fcntl.h> ... 8, 64
 - FD_CLOEXEC ... 15, 22, 74, 82
 - FIFO ... 64-65
 - File Advisory Information ... 63
 - file descriptor ... 14-16, 22-23, 63-65, 71-72, 74
 - Files and Directories ... 29
 - File Space Control ... 64
 - file system ... 65
 - pthread_join()*
 - with timeout ... 101
 - fork ... 13
 - fork()* ... 2, 23-25, 72-75, 77
 - fork handlers ... 24
 - Form of the Timeout Interfaces ... 101
 - free()* ... 68
 - ftruncate()* ... 65-66
 - functions
 - clock_getcpuclockid()* ... 49
 - mq_timedreceive()* ... 55
 - mq_timedsend()* ... 53
 - posix_fadvise()* ... 63
 - posix_fallocate()* ... 64
 - posix_madvise()* ... 66
 - posix_memalign()* ... 67

posix_spawn() ... 20
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_spawnp() ... 20
pthread_getcpuclockid() ... 50
pthread_mutex_timedlock() ... 35
sem_timedwait() ... 33

G

General ... 1
 General Concepts—measurement of execution
 time ... 6
 General Concepts ... 6, 71
 General Terms ... 5
 generate a signal ... 92
 Get Configurable Pathname Variables—
 Description ... 29
 Get Configurable Pathname Variables ... 29
 Get Configurable System Variables—
 Description ... 27
 Get Configurable System Variables ... 27
getitimer() ... 93
getrusage() ... 93

H

Headers and Function Prototypes ... 7
 Historical Documentation ... 69
 Historical Documentation and Introductory
 Texts ... 69
 History of Resolution Issues: Deletion of the
 enable attribute ... 94

I

IBM ... 94
 IEEE ... 75
 IEEE P1003.1a ... 74
 IEEE Std 1003.1 ... 5-6
 IEEE Std 1003.1b ... 6
 IEEE Std 1003.1c ... 6
 IEEE Std 1003.1d ... 6
 IEEE Std 1003.5 ... 6
 IEEE Std 1003.5b ... 6
 Implementation Conformance ... 2
 implementation defined ... 5-6, 19, 24, 39-
 40, 42-45, 48-49, 65, 71, 92-93
 Input and Output Primitives ... 31
 Introduction ... 91
 I/O Advisory Information and Space Control
 ... 63
 I/O Redirection with *posix_spawn()* ... 86
 ISO/IEC 14519 ... 6, 69, 73, 75
 ISO/IEC 9899:1995 ... 5
 ISO/IEC 9899 ... 5, 68
 ISO/IEC 9945-1 ... 5
 ISO/IEC 9945 ... 7

J

job control ... 74

K

kill() ... 25

L

language binding ... 72-73, 75-76
 Library-Level vs. Kernel-Level Implementa-
 tion ... 89
 Library Implementations of Threads ... 94
 Limitation of the Number of Pending Replen-
 ishments ... 90
 <limits.h> ... 8
lio_listio() ... 96
 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
... 35

Locking and Unlocking a Mutex—Errors
... 36

Locking and Unlocking a Mutex—Returns
... 36

Locking and Unlocking a Mutex—Synopsis
... 35

Locking and Unlocking a Mutex ... 35

login ... 77

M

malloc() ... 68

measurement of execution time ... 71
definition of ... 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() ... 67

MMU ... 73, 75

mq_open() ... 55, 57

mq_receive() ... 55, 96

mq_send() ... 53, 96

mq_timedreceive() ... 7, 55-56, 61
function definition ... 55

mq_timedsend() ... 7, 53-54, 61
function definition ... 53

<mqqueue.h> ... 8

mutexes ... 35

Mutexes ... 35

MVS ... 94

N

NULL ... 80-81, 83-84, 99-101

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 ... 9

options
Advisory Information ... 7-8, 10, 27, 29,
63-64, 66-67

Memory Mapped Files ... 66-67

Message Passing ... 7, 53, 55

Prioritized Input and Output ... 31

Process Scheduling ... 7, 18-19, 22-24,
31, 44

Process Sporadic Server ... 10, 27, 39-42

Semaphores ... 33

Shared Memory Objects ... 66-67

Spawn ... 7, 10, 14, 17, 19, 21, 27

Threads ... 5, 7, 24, 35

Thread Sporadic Server ... 10, 27, 39-40,
43-45

Timeouts ... 7, 10, 27, 33, 35-36, 53, 55

Timers ... 33, 54-55

options
Process CPU-Time Clocks ... 8, 10, 13, 27,
47-49, 59

Thread CPU-Time Clocks ... 7, 10, 13, 27,
48-50, 92

Other Standards ... 69

Overhead Considerations ... 92

P

package
POSIX_Process_Primitives ... 74

PATH
variable ... 22

pathconf() ... 9, 103-104

pathname ... 21

Pathname Variable Values ... 9

_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 ... 64-65, 77
- popen()* ... 77
- POSIX.1 ... 5, 8, 39, 47, 59, 75-76, 93-96
 - abbreviation ... 5
 - definition of ... 5
- 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, 72, 75
 - abbreviation ... 6
 - definition of ... 6
- _POSIX_ADVISORY_INFO ... 2, 10, 27, 63-64, 66-67
- POSIX_ALLOC_SIZE_MIN ... 9, 29, 103
- _POSIX_CPUTIME ... 2, 10-11, 13, 27, 47-49
- POSIX_FADV_DONTNEED ... 103
- posix_fadvise()* ... 7, 61, 63-65, 67, 103
 - function definition ... 63
- POSIX_FADV_NOREUSE ... 103
- POSIX_FADV_RANDOM ... 103
- POSIX_FADV_SEQUENTIAL ... 103
- POSIX_FADV_WILLNEED ... 103
- posix_fallocate()* ... 7, 61, 64-65, 103
 - function definition ... 64
- POSIX_MADV_DONTNEED ... 103
- posix_madvise()* ... 7, 61, 64, 66-67, 103
 - function definition ... 66
- POSIX_MADV_RANDOM ... 103
- POSIX_MADV_SEQUENTIAL ... 103
- POSIX_MADV_WILLNEED ... 103
- _POSIX_MAPPED_FILES ... 66
- posix_memalign()* ... 8, 67-68, 104
 - function definition ... 67
- _POSIX_MESSAGE_PASSING ... 53, 55
- _POSIX_PRIORITIZED_IO ... 31
- _POSIX_PRIORITY_SCHEDULING ... 11, 19, 22-24, 31, 44
- POSIX_Process_Primitives
 - package ... 74
- POSIX_REC_INCR_XFER_SIZE ... 9, 29, 104
- POSIX_REC_MAX_XFER_SIZE ... 9, 29, 104
- POSIX_REC_MIN_XFER_SIZE ... 9, 29, 104
- POSIX_REC_XFER_ALIGN ... 9, 29, 103-104
- _POSIX_SEMAPHORES ... 33
- _POSIX_SHARED_MEMORY_OBJECTS ... 66
- posix_spawn()* ... 7, 14-16, 18, 20-21, 24-25, 61, 71-77, 86
 - Equivalent ... 86
 - function definition ... 20
- _POSIX_SPAWN ... 2, 10, 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()* ... 7, 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, 14-15, 25
 function definition ... 14

posix_spawnnp() ... 7, 14-16, 18, 20-22, 24-25, 61, 71-77, 86
 function definition ... 20

POSIX_SPAWN_RESETEIDS ... 18, 23, 72, 78, 80

POSIX_SPAWN_SETPGROUP ... 18, 22, 24, 78, 80

POSIX_SPAWN_SETSCHEDPARAM ... 18-19, 22-24, 78, 82

POSIX_SPAWN_SETSCHEDULER ... 18-19, 22-24, 78, 82

POSIX_SPAWN_SETSIGDEF ... 18, 23, 78, 80

POSIX_SPAWN_SETSIGMASK ... 18, 23, 78, 80

_POSIX_SPARADIC_SERVER ... 2, 10-11, 27, 39-42

_POSIX_SS_REPL_MAX ... 8-9, 90

_POSIX_THREAD_CPUTIME ... 2, 10-11, 13, 27, 47-50, 59

_POSIX_THREAD_PRIORITY_SCHEDULING ... 11, 43

_POSIX_THREADS ... 35

_POSIX_THREAD_SPARADIC_SERVER ... 2, 10-11, 27, 39-40, 43-45

_POSIX_TIMEOUTS ... 2, 10, 27, 33, 35, 53, 55

_POSIX_TIMERS ... 11

PRIO_INHERIT ... 36

Prioritized Input and Output option ... 31
 procedure
 Start_Process ... 72-76
 Start_Process_Search ... 73, 75

Process CPU-Time Clocks option ... 8, 10, 13, 27, 47-49, 59

Process Creation—Description ... 13

Process Creation ... 13

Process Creation and Execution ... 13-14, 71

Process Environment ... 27

process group ... 18, 22, 25, 74, 76

process group ID ... 22, 74

process ID ... 22, 24

Process Primitives ... 13, 71

Process Scheduling Attributes ... 43

Process Scheduling Functions ... 41

Process Scheduling option ... 7, 18-19, 22-24, 31, 44

Process Sporadic Server option ... 10, 27, 39-42

Process Termination ... 25

pthread_cond_signal() ... 97

pthread_cond_timedwait() ... 96-97, 99, 101

pthread_cond_wait() ... 96, 99

pthread_getcpuclockid() ... 7, 47, 50-51, 92, 94
 function definition ... 50

<pthread.h> ... 8

pthread_join() ... 96, 99

pthread_mutex_lock() ... 35, 96-97

pthread_mutex_timedlock() ... 7, 35-36, 97-98
 function definition ... 35

PTHREAD_ONCE_INIT ... 99

pthread_setschedparam() ... 45

pthread_timedjoin() ... 99

R

Range of Scheduling Priorities ... 90

read() ... 96

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, 90

replenishment period ... 39-41, 87-88, 90

Requirements for Timeouts ... 95

resolution
 clock ... 33, 36, 54, 56

Run-Time Invariant Values (Possibly Indeterminate) ... 8-9

running thread ... 40-41

S

_SC_ADVISORY_INFO ... 27
 limit definition ... 27

_SC_CPUTIME ... 27
 limit definition ... 27

SCHED_FIFO ... 31, 35, 39-40, 42-44, 90, 97

- sched_get_priority_max()* ... 41
- sched_get_priority_min()* ... 41
- <sched.h> ... 39
- SCHED_OTHER ... 39, 42, 44
- schedparam
 - attribute ... 43
- SCHED_RR ... 31, 35, 39, 42-44, 97
- sched_setparam()* ... 24-25
- sched_setscheduler()* ... 24-25
- SCHED_SPORADIC ... 31, 35, 39-45, 90
- Scheduling Allocation Domain ... 43-44
- Scheduling Aperiodic Activities (rationale) ... 88
- Scheduling Documentation ... 44
- Scheduling Parameters ... 39
- Scheduling Policies ... 39-40
- scheduling policy ... 35, 39-43, 45, 89-90, 97
- Scope ... 1
- _SC_PAGESIZE ... 66-67
- _SC_SPAWN ... 27
 - limit definition ... 27
- _SC_SPORADIC_SERVER ... 27
 - limit definition ... 27
- _SC_THREAD_CPUTIME ... 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, 96
- 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()* ... 93
- setpgid()* ... 24-25, 76
- Set Scheduling Parameters—Description ... 41-42
- Set Scheduling Parameters ... 41
- Set Scheduling Policy and Scheduling Parameters—Description ... 42
- Set Scheduling Policy and Scheduling Parameters ... 42
- setuid()* ... 25
- Shared Memory Objects option ... 66-67
- shell ... 77
- shell ... 77
- SIG_DFL ... 23, 80
- SIG_IGN ... 23
- signal
 - generate ... 92
- signal actions ... 23
- signal mask ... 23
- SIG_SETMASK ... 80
- sigtimedwait()* ... 96, 101
- sigwait()* ... 96
- Spawn a Process ... 20, 73
- Spawn Attributes ... 16, 73
- spawn-default
 - attribute ... 18
- Spawn File Actions ... 14, 71
- spawn-flags
 - attribute ... 18, 20, 22-24
- <spawn.h> ... 8, 14, 17-18, 22
 - header definition ... 14
- Spawning a new Userid Process ... 87
- spawn option ... 19
- Spawn option ... 7, 10, 14, 17, 21, 27
- spawn-pgroup
 - attribute ... 18, 20, 22, 76
- spawn-schedparam
 - attribute ... 19-20, 23
- spawn-schedpolicy
 - attribute ... 19-20, 23
- spawn-sigdefault
 - attribute ... 19-20, 23
- spawn-sigmask
 - attribute ... 18, 20, 23
- Spinlock Implementation ... 97
- Sporadic Server Scheduling Policy ... 87
- SS_REPL_MAX ... 9, 42-45, 90
- Start_Process
 - procedure ... 72-76
- Start_Process_Search
 - procedure ... 73, 75
- stat()* ... 25
- <stdlib.h> ... 8
- Symbolic Constants ... 10
- Synchronization ... 33

sysconf() ... 9, 66-67
 <sys/mman.h> ... 67
system() ... 72, 74, 77
 system clock ... 33, 36, 54, 56
 System V ... 27, 94

T

Terminology and General Requirements ... 5
 terms ... 5
 Thread Cancellation ... 61
 Thread Cancellation Overview ... 61
 Thread CPU-Time Clocks option ... 7, 10, 13, 27, 48-50, 92
 Thread Creation—Description ... 59
 Thread Creation ... 59
 Thread Creation Scheduling Attributes—Description ... 44
 Thread Creation Scheduling Attributes ... 44
 Thread Functions ... 59
 Thread Management ... 59
 Thread Scheduling ... 43
 Thread Scheduling Attributes ... 43
 Thread Scheduling Functions ... 44
 Threads option ... 5, 7, 24, 35
 Thread Sporadic Server option ... 10, 27, 39-40, 43-45
time() ... 33-34, 36-37, 54-57
 <time.h> ... 8, 34, 36-37, 49, 54-57
 Timeouts option ... 7, 10, 27, 33, 35-36, 53, 55
timer_create() ... 48, 50-51
 Timers option ... 33, 54-55
times() ... 25, 93-94
 TOC ... 5
 TRUE ... 98, 101

U

undefined ... 14-15, 18, 40
 UNIX ... 74-75, 93
unlink() ... 66
 Unlock a Semaphore—Description ... 35
 Unlock a Semaphore ... 35
 unspecified ... 18-19, 24-25, 31

V

Versioned Compile-Time Symbolic Constants ... 11

W

wait() ... 25, 76-77
 Wait for Process Termination — Description ... 25
 Wait for Process Termination ... 25
waitpid() ... 25, 76-77
 WEXITSTATUS ... 77
 Which Services Should Be Timed Out? ... 95
 WIFEXITED ... 25, 77
 WIFSIGNALED ... 25
 WIFSPAWNFAIL ... 76
 WIFSTOPPED ... 25
write() ... 96
 WSPAWNERRNO ... 76
 WSTOPSIG ... 25