POSIX Threads
Catalog of Portability Issues

Prepared by

Offer Pazy [1]

under contract to

Florida State University
Tallahassee, FL 32306-4019

August 5, 1994

for

HQ U.S. Army CECOM, Software Engineering Directorate [2]

via

TELOS SYSTEMS GROUP
55 North Gilbert Street
Shrewsbury, NJ 07702

for delivery under

Purchase Order No. TSG-2056, Modification 1
Prime Contract: DAAB07-91-D-Q502/D.O. 0017

Copyright © 1994 Florida State University.
All rights reserved.

---

[1] The author would like to thank Ted Baker and Ted Giering of Florida State University for the numerous discussions we have held and the reviews they have provided. This help was instrumental for the development of this document and improved its quality substantially. However, the author is still responsible for the technical content and opinions expressed in this document.

[2] The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

Contact Information

The author's present electronic address is pazy@world.std.com. The document is maintained by the PART Project at Florida State University. Questions, problem reports, and comments on how this document might be improved are very welcome. All correspondence should be forwarded to:

Prof. Ted Baker
Department of Computer Science (4019)
Florida State University
Tallahassee, FL 32306-4019
USA

phone: (904) 644-5452
e-mail: baker@cs.fsu.edu

(Electronic mail is preferred.)

Permissions

This is a Florida State University technical report. The publication, distribution, or copying of this draft, as well as all derivative works based on this draft, is expressly prohibited except as set forth below.

Permission is hereby granted to everyone to reproduce and distribute this document in its entirety, subject to the restrictions contained herein.

If this document is reproduced or transmitted in any form, all copies or portions thereof must include the original title page, including the Florida State University copyright notice with credits and disclaimer, and this entire Permissions notice in a prominent location.

Draft 0 Original draft by Offer Pazy, distributed for review at the POSIX meeting, July 1994.

Draft 1 Edited by T.P. Baker, in response to comments by Mary Bender, August 1994.

Abstract

This document collects and classifies the explicit and implicit implementation dependences that are present in the proposed POSIX Threads Extension, P1003.1c (originally named P1003.4a). Dependences that are carried over from IEEE-STD 1003.1–1990 and IEEE-STD 1003.1b–1993 are not discussed. Only semantic issues at the interface level are discussed. Consequently, the analysis of the many valid implementation approaches is considered out of scope for this report. The focus is on issues that affect application developers; some discussion on language bindings and system software development is also presented. It is argued that too many such implementation dependences exist for the standard to be useful for demanding applications. Since P1003.1c is still in a draft form at the time of completing this report, we hope that our findings will help improve the standard before its final approval.

Chapter 1: Introduction

1.1 Purpose

The purpose of this document is to collect and list all instances in P1003.4a[1] {B2} where the required behavior is not fully defined, and the actual semantics are left to the implementation. Several categories of such implementation dependences are introduced and each instance is classified accordingly.

This document covers only issues that are introduced by P1003.4a or by its intended integration with the rest of the POSIX standard Application Program Interfaces (APIs) (namely, POSIX.1{B1} and POSIX.4{B3}). Clearly, an application developer needs to know all implementation-dependences whether they are introduced by P1003.4a or by any of the other parts of POSIX. However, the effort to develop such a comprehensive list and the size of the resulting document is beyond the scope of this task.

The main focus of this document is on issues of portability that may affect C-language application users of the standard. In addition, issues that affect developers of an Ada RTS (specifically, Ada 9X) on top of P1003.4a are discussed. Finally, issues that are significant to the language bindings to P1003.4a (specifically, Ada bindings) are analyzed.

Note that this document does not cover issues that are internal to the implementation (although such issues are quite important in their own right). It concentrates only on the semantic aspects of the interface. As a result, pure implementation choices that affect trade-offs, such as space vs. time, speed vs. time-boundedness, and safety vs. speed, are not discussed. This decision is based on the scope of this document, the variety of user needs, and the large number of possible implementation approaches. (For example, a long-running application that must also be fault-tolerant has a completely different set of requirements than one that is very small and runs for a very short time.)

Finally, we hope that the timing of the distribution of this document will help the POSIX working group improve the P1003.4a document and fix the remaining problems before the standard's final approval.

1.2 Structure

The structure of this report follows roughly that of the P1003.4a document. After this introduction, we define, in chapter 2, the various terms and conventions used throughout this document. In chapter 3, we itemize those issues that we believe are the most problematic with respect to portability. Chapter 4 discusses issues that relate to P1003.4a as a whole; Chapters 5 through 18 discuss issues specific to individual chapters. Chapter 19 summarizes the findings of this document and draws some conclusions.

[1] The original name P1003.4a was changed by the IEEE to P1003.1c, between Draft 8 and Draft 9.

For each entry, we provide its category, an exact reference to the draft standard, and a short description of the issue. A discussion follows when either the nature of the problem and/or its effects deserve more elaboration.

Unless otherwise specified, all the references are to P1003.4a. At the time of writing this report (June 1994), the latest draft is Draft 9. This is only a list of changes to Draft 8, which is the last draft to be printed as a full document. Due to the incomplete nature of Draft 9, for specific lines of text we need to make references to both Draft 8 and Draft 9. To keep it clear to which draft a line reference applies, the draft number is specified, in parentheses.

Chapter 2: Terms and Conventions

2.1 Categories

Below we describe the taxonomy of implementation dependences used throughout this document. All of the entries are classified according to the following categories: (Where quotations appear, they are from POSIX.1.)

[ID]
: *"An indication that the implementation shall define and document the requirements for correct program constructs and correct data of a value or behavior."*

[UND]
: *"An indication that this part of ISO/IEC 9945 imposes no portability requirements on an application's use of an indeterminate value or its behavior with erroneous program constructs or erroneous data. Implementations (or other standards) may specify the result of using that value or causing that behavior. An application using such behaviors is using extensions, as defined in 1.3.2.3."*

[UNS]
: *"An indication that this part of ISO/IEC 9945 imposes no portability requirements on applications for correct program constructs or correct data regarding a value or behavior. Implementations (or other standards) may specify the result of using that value or causing that behavior. An application requiring a specific behavior, rather than tolerating any behavior when using that functionality, is using extensions, as defined in 1.3.2.3."*

[MAY]
: *"An indication of an optional feature. With respect to implementations, the word* may *is to be interpreted as an optional feature that is not required in this part of ISO/IEC 9945, but can be provided."* Note that POSIX.1 does not require the implementation to document its choices. *"The conformance document may specify the behavior of the implementation for those features where this part of ISO/IEC 9945 states that implementations may vary ..."* (there 1.3.1.2, lines 80-82).

[OPT]
: A designated option of the standard.

[IMP]
: An implicit dependence on implementation behavior.

[HOL]
: A hidden hole or missing semantics in the standard.

[ICON]
: A conflict arising when integrating POSIX.4 and P1003.4a.

[IHOL]
A semantic hole when integrating POSIX.4 and P1003.4a.

[ERR]
A "simple" user error situation. It is only of minor interest for the purposes of this document since such usage is considered "pathological" and should be avoided by all applications not just portable ones. Unless something special has to be said for a particular entry, no discussion is presented. However, the differences in implementation behavior may affect the coding of error recovery routines and the reliability of the application.

2.2 Terminology for Priorities

Throughout this document, we use the term *base priority* for the priority with which a thread is created (or which is later modified by *pthread_setschedparam()*), and active priority for the effective scheduling priority of the thread at any instant. The difference is that the active priority includes the effects of any adjustments that have been made as a result of mutexes according to various inheritance protocols.

The POSIX documents do not explicitly recognize that there are two kinds of priorities. Instead, in various places, the text only implies that such a distinction exists and provides special, "local" rules for it. This incomplete approach is apparent in the terminology. Often, the term "priority" is used without any qualifier, which makes the interpretation of the kind of priority being discussed difficult since this interpretation must be derived from the context. In other instances, qualifiers such as "original" (13.6.1.2 [LINES: 542 (D9)]), "temporary" (13.5.2.2 [LINES: 433 (D8)]), "current" (POSIX.4 6.7.1.1 [LINES: 395 (D8)]), "execution" (13.6.1.2 [LINES: 621-622 (D8)] and POSIX.4 13.1 [LINES: 9 (D8)]), or "inherited" (13.6.1.2 [LINES: 621-622 (D8)]) are used in a way that makes it difficult to tell whether these are formal technical terms. This would suggest that when the word "priority" appears by itself the intent is different. To conclude, one has to make his own determination about the kind of priority being referred to in cases where no qualifier is given and whether to treat terms like "execution priority" and "temporary priority" as formal technical terms.

Chapter 3: Critical Portability Issues

3.1 Introduction

The most immediately apparent result of this documentation effort is the large number of issues that were identified. While each issue can be addressed on its own and an adequate work-around can be found, the fact that so many such implementation-dependences exist, makes writing a completely portable application (that runs on every conforming implementation without any changes)almost unattainable. Several of the implementation-dependences can be isolated by careful coding, since the effect of such dependences will be local. For others, the dependences affect the high-level design of the application, and as such have much more serious implications. Experience has shown that the quantity (of portability issues that a programmer must consider) definitely has a qualitative effect on the development effort and the resulting software.

In analyzing the portability issues of P1003.4a, we have been very careful to rely only on the normative text. The rationale occasionally provides clarifications as to the original intent (when this is not clear in the normative text). Also, several of the technical reviewers provided us with interpretations, and we provided many other answers through educated guesswork. However, we do not believe that this is good enough. For POSIX to be a successful standard, one that is in wide-spread use, it must be self-contained and fully consistent internally, so that any reader, be it a user or an implementer, can completely understand the rules without the help of the original developers.

Note that many entries are classified as [ERR]. Most of these cases, we believe, are not a sigificant portability concern, except for applications that need to be fault tolerant, or at least provide some form of local error recovery.

The subject of developing Ada RTSs (mainly for Ada 9X) on top of a P1003.4a implementation deserves special attention. The question of whether or not language run-time systems are in the scope of P1003.4a was very controversial during the development of the draft. It seems that the current consensus is that such a goal is indeed out of scope. We disagree with this position, however the purpose of frequently mentioning Ada and Ada RTSs throughout this document is not to reopen the debate. Rather, we believe that many issues encountered while developing an RTS will come up when designing any complex, system-level application that is required to do its own scheduling or multiplexing and respond to external events. A good example would be a database server. The suitability of P1003.4a as an execution platform for the development of an Ada RTS was studied extensively during the past several years {B6, 7, 8, 9}. On the other hand, we are not aware of any published reports on the corresponding subject of using P1003.4a for system-level applications. Therefore, the comments about P1003.4a and Ada should be evaluated in the broader scope as a possible indication of the problems that will affect, in the future, a larger audience.

The semantic rules of Ada are very specific, with very little room for implementation dependences. Therefore, the developer of an Ada RTS faces a dilemma: Either use the underlying P1003.4a and all of its features, (but then be subject to all its implementation dependences and the need to work around them); or ignore most of these features and build everything from scratch. In addition to the development cost associated with such a decision, it also has a major influence on the interoperability of applications using this Ada RTS. It is important to note that a large number of these issues have to be addressed in the design phase of the Ada RTS; it cannot be left to be solved during the low-level coding. In essence, the designer has to assume the worst about the underlying P1003.4a implementation (in terms of making the appropriate implementation choices), and to design the Ada RTS using the lowest common denominator of all P1003.4a implementations.

Finally, GNARL, the Ada RTS for GNAT{B5} has shown, that an Ada 9X RTS can, in fact, be implemented on top of P1003.4a. However, it is yet too early to determine whether or not this RTS is complete and validatable. In addition, no performance figures are known at this point. Finally, the underlying P1003.4a implementation for this effort is a home-grown implementation which is based on an early draft of P1003.4a. One characteristic of this implementation is that all operations are async-signal-safe, something that is not required by P1003.4a. Whether the result of this effort can be used effectively on a fully-conformant, but non-async-safe P1003.4a implementation, is yet to be determined. One fact that is already clear from this effort is that while there seem to be solutions for all the problems facing an Ada RTS development, the design choices of the developer are limited substantially by the limitations of P1003.4a, the many uncertainties in the interpretation of the semantics, and the large number of implementation dependences. In the opinion of the author, this left GNARL with only one feasible (and not necessarily very efficient) approach for the overall design. 1

3.2 Most Serious Problems

We realize that the issues discussed in this report vary greatly in their importance and relevance. Some are critical, others may be important to some users, and still others may be so trivial that they are unlikely to be of interest to application developers. Therefore, we would like to draw special attention to the following items which, in our judgement, are the most serious and have the greatest effect on portability. The order in which these issues are listed does not reflect any sort of ranking [1].

— Which options are supported, out of the many independent options into which the proposed standard is divided? See 5.4 for details.

— What is the effect on a blocked thread of signal delivery to that thread? See 6.16 for sigwait functions, 11.2 for semaphore waits, and 11.16 for condition variable waits.

— What is the effect on a blocked thread of signal delivery to another thread? See 6.6 for details.

[1] This list is not intended to imply that these are the only eleven important issues. Moving down the scale of importance, we were forced to draw a line, at a fairly arbitrary point. There are other serious issues, discussed in the body of this report, some of which could actually be more critical for specific applications.

— Can *pthread_cond_signal()* wake up more than one thread? (i.e. are spurious wake-ups allowed?) See 11.11 for details.

— What is the meaning of the priority of a process where threads are supported? See 13.8 for details.

— Which priority is used for the queuing operations? The base priority or the active one? See 13.3 for details.

— What are the interactions between dynamic priority changes and other priority inheritance rules? See 13.4 for details.

— Which contention scopes are actually supported? See 13.12 for details.

— What are the allocation domains in a given system, and how does an application find out (or specify) this? See 13.13 for details.

— How well is the development of fault tolerant applications supported? See 4.5 for details.

— What are the (implementation-defined) thread cancellation points? See 18.3 for details.

Chapter 4: Non Section-Specific Issues

4.1 Introduction

This chapter covers issues that are either common to several sections in the POSIX document, or that do not refer to an exact section of P1003.4a.

4.2 Permissions Needed for Setting Scheduling Parameters

Class: [ID]

Reference: 13.5.2.4 [LINES: 470-472 (D8)]

Reference: 13.6.1.4 [LINES: 651 (D8)]

Reference: 13.6.2.4 [LINES: 734 (D8)]

Issue:

Whether special permissions are required to set the scheduling parameters.

Discussion:

POSIX APIs allow certain operations to be governed by appropriate administrative privileges. The methods for obtaining these privileges are outside the scope of POSIX.1, POSIX.4, or P1003.4a. The specific issue here is the effect of this administrative issue on the scheduling and timing of real-time applications. Specific questions are:

— Are the rules different for threads in the same process, and for threads in other processes (or in process groups)?
— Does the answer differ for different scheduling parameters?

This is, of course, a major issue for any application that wishes to set those parameters dynamically. While the consequence of this implementation dependence is that the above capability is not portable, in practice, setting the desired permissions can be done once, presumably at the start of the application execution. Thus, the effect of this non-portable code will be quite local. However, this issue does mean that, in general, one cannot rely on this feature (that is, the dynamic setting of scheduling parameters) to be supported by every POSIX.1 environment, even if it supports the _POSIX_THREAD-_PRIORITY_SCHEDULING option.

Note that the same is not true for the operations that set the thread creation scheduling attributes (13.5.1), although setting these attributes affects the scheduling parameters of a thread. P1003.4a does not allow the rejection of these operations (or the *pthread-_create()* function) with [EPERM] due to the lack of the appropriate permissions.

4.3 Concurrent Access to Shared Resources

Class: [IHOL]

Issue:

What is the effect of concurrent access to shared resources on the implementation?

Discussion:

This is the general issue of operating on resources, particularly files, from multiple threads without coordination. Clearly, the resource itself may be corrupted. The real question is: Are there any limits on the erroneous behavior of the implementation?

4.4 Use of "Virtual Processors"

Class: [IMP]

Issue:

The effect of employing "virtual processors" on potential deadlocks.

Discussion:

Virtual processors [a.k.a "kernel entities" or "light-weight-processes" (LWP)] are emerging as a popular approach to threads implementation. The idea is that a small number of such entities is allocated by the kernel, and the library portion of the implementation multiplexes them among the user threads. This is a promising approach that addresses many of the trade-offs between kernel and library implementations.

When a user thread is blocked on a kernel call (such as "read"), its current virtual processor (VP) is taken out from the pool of available VPs. If the number of such VPs is statically allocated, a deadlock may occur when enough user threads are blocked waiting for other threads to make process, while these other threads do not have a VP to execute on. Note that the source of such a deadlock is not the application, which in fact cannot do anything about it.

At least two possible solutions exist: One is to dynamically allocate more VPs in this case. The other is to return some sort of "capacity-error" indication when the thread on the last VP is about to block.

It is our belief that the implementation is non-compliant if it allows such deadlocks to occur (see 2.3.7). So formally, this is not an issue of portability. However since such a bug is intermittent, it will be very hard to detect while validating the implementation or testing a given application. This makes the problem for the user much more serious.

A related question is whether it is permissible for an implementation on a single-processor system to define allocation-domains (see 13.4.3) larger than one, using LWPs instead of processors. There is no language in P1003.4a that prohibits such behavior. It means that, even on a single-processor system, a user cannot count on the allocation domain to always equal one, and (as is discussed in 13.14), for allocation domains larger than one, the scheduling rules are entirely implementation defined.

4.5 Support for Fault Tolerant Applications

Class: [IMP]

Issue:

How well does P1003.4a support the development of fault tolerant applications.

Discussion:

The position of the current P1003.4a draft is that since threads share memory, they cannot be used as the unit of encapsulation for error detection, containment, and recovery. Instead, processes should be used. This view is inconsistent with the model of the two small realtime-system profiles of {B4}, which do not have support for multiple processes.

This issue comes up in various sections of the P1003.4a document where certain conditions are treated as erroneous and therefore, their precise semantics are not defined. Obviously, a portable, fault tolerant application must know the precise rules in those cases. Most of the entries classified as [ERR]are relevant to such applications. In addition, see 4.4, 4.6, 5.1, 15.1, 16.1, 16.2, 16.3, 16.8, 16.11, 18.3, 18.5, and 18.7 for other related issues.

4.6 C-language Programming Errors

Class: [ERR]

Issue:

The effect of calling P1003.4a functions with erroneous parameters.

Discussion:

This is a general C-language issue originating from the fact that C-language is not a strongly-typed language. However, since P1003.4a is not implemented by the application developer, there is an added interest in finding how such an implementation will react to user errors. We should note that this is not often an issue of the P1003.4a implementation, but rather of the C-language compiler. It is also often the case that the P1003.4a implementation cannot "see" certain user errors (such a specifying a pointer to the wrong structure.)

Examples:

— The result of having the character array pointed to by *name* shorter than *namesize* (4.2.4.2, [LINES: 10-12 (D8)] and 4.7.4.2 [LINES: 63-64 (D8)]).

— What happens if the input parameters to *strtok_r()* do not follow the rules specified in this section? (8.3.4.2 [LINES: 201-215 (D8)]).

— What happens if *buf* on calls to *asctime_r()* or *ctime_r()* is less than 26 characters? (8.3.5.2 [LINES: 241-244 (D8)] and 8.3.6.2 [LINES: 270-272 (D8)], respectively).

4.7 Implementation-Defined Errors

Class: [MAY]

Issue:

What are the implementation-defined error conditions?

Discussion:

POSIX.1 (in 2.4), gives a general permission for the implementation to define and return errors other than the ones that are defined by the standard or to return the errors defined by the standard under circumstances not described by the standard. In addition, many of the errors (marked by an *"if detected"* clause) need not be detected.

Naturally, this information is very important for the reliability and error-recovery of applications. It is unclear how a portable application can be developed where all the error conditions are essentially implementation defined. However, this issue is the legacy of POSIX.1 and it is not a new one introduced by P1003.4a.

Chapter 5: Definitions

This section describes portability issues related to Section 2 of P1003.4a.

5.1 Memory Synchronization After Failed Operations

Class: [UNS]

Reference: 2.3.8 [LINES: 402-403 (D8)]

Issue:

If an operation defined in this list returns an error, is the memory synchronized?

Discussion:

This paragraph requires the implementation to synchronize memory after the normal completion of any function included in the list provided there. If the given function fails, P1003.4a leaves the behavior unspecified. This is a subtle issue that is not likely to affect many users, since it is quite difficult to rely on the behavior of failing functions.

5.2 Implementation-Defined Thread-Safe Functions

Class: [ERR], [MAY]

Reference: 2.3.9

Issue:

This paragraph lists a number of functions that need not be “thread-safe”. Are these functions indeed not “thread-safe”, on a given implementation?

Discussion:

This issue is not perceived as important, since thread-safe versions exist for these functions. It is basically a user error to use the non-thread-safe versions in a multi-threaded application.

5.3 Minimum Values

Class: [OPT]

Reference: 2.8.4

Issue:

The maximum number of thread-specific data keys, maximum number of threads per process, and the number of attempts to destroy a thread-specific data value, are implementation-defined.

Discussion:

The significance of these values depends on whether or not the minimum portable values are "sensible". They are sensible for an average application, but may be too small for a large application or for an Ada RTS (which usually needs to support applications of different sizes).

5.4 Optional Functionality

Class: [OPT]

Reference: 2.9.3 [LINES: 710-726 (D9)]

Issue:

The support for threads in P1003.4a is divided into many independent options.

Discussion:

This is a major concern which may be addressed adequately by future profiles, but so far the development of these is lagging behind. Since there is no minimal required functionality, both C-language application developers and Ada RTS builders face a problem in the design phase. They have to decide on which underlying P1003.4a functionality they depend. Note that, in most cases, if the code depends on a given option and that option is not present, a substantial redesign has to be carried-out (as opposed to a local re-coding).

For a language-binding such as Ada, this division into optional parts presents a difficulty since it is not always feasible to represent it using Ada packages.

Chapter 6: Signals

This section describes portability issues related to Section 3.3 of P1003.4a.

6.1 Missing Semantics for the Notification Thread

Class: [HOL]

Reference: 3.3.1.2 [LINES: 153-165 (D9)]

Issue:

What are the default attributes of a new thread created by a notification?

Discussion:

When *sigev_notify_attributes* equals NULL, what are the values of the thread creation attributes such as stack-size, and scheduling parameters?

6.2 Creating a Joinable Notification Thread

Class: [ERR], [UND]

Reference: 3.3.1.2 [LINES: 162-164 (D9)]

Issue:

What is the effect of the creation of a notification thread with a *detachstate* attribute equal to PTHREAD_CREATE_JOINABLE?

6.3 The Signal Mask of a Notification Thread

Class: [ID]

Reference: 3.3.1.2 [LINES: 164-165 (D9)]

Issue:

The signal mask for a thread created by the notification.

Discussion:

This is of course a major issue since it influences the very first operation of such a thread. (This issue is separated from 6.1 since it is explicitly implementation-defined.)

6.4 Definition of Async-Signal-Safety

Class: [ICON]

Reference: 3.3.1.3

Issue:

Inconsistencies between POSIX.1 and P1003.4a with respect to async-signal-safety.

Discussion:

There is a slight difference in terminology between P1003.4a and POSIX.1, with regard to async-signal-safety. POSIX.1 (in 3(f)) specifies that certain operations are *unsafe* with respect to signals. The only situation in which these operations are unsafe to call is when they are called from a signal handler and the signal interrupted another unsafe operation. P1003.4a (in 2.2.2.14) defines *async-signal-safe* to mean a function that *"may be safely invoked, without restriction, from signal-catching functions"*. This definition refers to 3.3.1.3. We deduce that the unsafe operations of 3.3.1.3 are exactly the complement of the async-signal-safe operations of 2.2.2.14.

There remains a slight ambiguity in 3.3.1.3, where it uses the phrase *"when a signal interrupts an unsafe function"*. What does it mean for a signal "interrupt a function" in a multithreaded environment? In particular, if a thread A is executing a function call and it is interrupted (preempted) by the operating system to deliver a signal to a different thread, B, was the function call by thread A interrupted by the signal? We infer this is not intended. That is a signal is said to interrupt a function call only if the function call is made by the same thread as executes the signal handler. However, the language of P1003.4a seems to leave room for an implementation to take the other point of view, and if it did, there would be a significant portability problem.

6.5 Additional Async-Signal-Safe Functions?

Class: [UNS]

Reference: 3.3.1.3 [LINES: 341-345 (D8)]

Issue:

No async-signal-safe functions are defined by P1003.4a.

Discussion:

Since the async-signal-safe property is quite important, it is likely that more functions will have it in a given implementation. This will create a portability problem, since whether or not one can use such a function in a signal handler is usually a design issue with non-local effect.

6.6 Signal Delivery and Blocked Threads

Class: [IMP]

Reference: 3.3.1.4

Issue:

The effect of signals delivered to a given thread on other blocked threads.

Discussion:

POSIX.1 says that a signal delivered to a process may interrupt a waiting function. P1003.4a defines some rules regarding how signals delivered to a process are directed to a specific thread. On the face of it, there is no permission to interrupt any arbitrary thread when a signal is delivered to the process. However, this is not entirely clear in the text. In addition, for library implementations of P1003.4a, such a permission is almost a requirement, since at the time the signal is delivered (from the implementation to the process), all threads may be blocked on a POSIX.1 system call. The implementation will have to interrupt one of these threads in order to deliver the signal to the process (i.e. to the P1003.4a library implementation). It is not clear that the P1003.4a library implementation can then issue the system call again on behalf of the interrupted thread. Therefore it may need to use [EINTR] instead.

6.7 Using an Uninitialized Signal Mask

Class: [ERR], [UND]

Reference: 3.3.3.2 [LINES: 394 (D8)]

Issue:

Using an uninitialized signal set for the signal set operations.

6.8 Using Sigwait Functions and *sigaction()* on the Same Signal

Class: [UNS]

Reference: 3.3.4.2 [LINES: 176-177 (D9)]

Reference: 3.3.8.2 [LINES: 215-216 (D9)]

Issue:

Using *sigaction()* and a sigwait function on the same signal.

Discussion:

The effect of a sigwait function on the signal actions is unspecified. This means that the action may be changed to ignore, the default, or set to some unknown handler, on return from the sigwait function.

In general, mixing *sigaction()* with a sigwait function on the same signal should be avoided. When needed, a portable application must ensure that the given signal is blocked when a sigwait function is called. It must not unblock it until *sigaction()* has

been called to establish a new action. However, when incorporating third-party code or when dealing with multi-mode applications with tight timing, such a disciplined used cannot be always guaranteed.

Language bindings must avoid exporting any combination of functions that may lead indirectly to such a simultaneous use, since the language binding cannot control the usage-pattern of such functions.

6.9 *sigprocmask()* in a Multi-Threaded Application

Class: [UNS]

Reference: 3.3.5.2 [LINES: 419 (D8)]

Issue:

Using *sigprocmask()* in a multi-threaded application.

Discussion:

In theory, this is an error situation since *sigprocmask()* should not be used in a multi-threaded application. However, in practice, this problem may affect some applications. For example, if an application is written for a system where threads are not supported, it must use *sigprocmask()* (since the equivalent function *pthread_sigmask()* is not defined). When this application then runs on a system where threads are supported (even if not actually being used), *sigprocmask()* becomes unspecified (based on the P1003.4a amendment).

6.10 *sigpending()*: Process' vs. Thread's Signal Set

Class: [IMP]

Reference: 3.3.6.2 [LINES: 446-448 (D8)]

Issue:

Which set of pending signals is being stored for *sigpending()*? The process' or the thread's?

Discussion:

This is largely a wording issue. The language used by P1003.4a: *"... and are pending either for the process or the calling thread."* is somewhat ambiguous. The sentence may be interpreted either as providing an implementation freedom to return one or the other, or as requiring the return of a union of the sets. If the latter interpretation is adopted, there is no portability issue. On the other hand, if the first interpretation holds, there is a portability concern, although not a major one in most cases. Whether a pending signal is considered pending for the process or for the thread, is essentially an issue of timing.

6.11 Meaning of System's Set of Signals

Class: [HOL]

Reference: 3.3.8.2 [LINES: 204 (D9)]

Issue:

What is the *"system's"* set of signals?

Discussion:

This seems to only be a clarification issue. It is either the process' or the thread's set of signals, depending on the situation.

6.12 Effect of a Sigwait Function on Pending Signals

Class: [ID]

Reference: 3.3.8.2 [LINES: 205-211 (D9)]

Issue:

Whether a sigwait function clears all pending signals of that number.

Discussion:

This is a major issue relevant to anyone developing a multiple component system or incorporating third-party software. If the application has tight timing constraints, a queue of signals may develop (note that real-time signals support queues). It is also unclear whether [LINES: 209-211 (D9)] (*"If the implementation supports queued signals ... "*) negate this implementation-dependency for real-time signals.

To protect this from occurring, all signal handlers (of the same signal) must be unified and must ensure (usually by incorporating a two-stage mechanism) that queues never develop.

Naturally, this is of major concern to an Ada RTS implementation since it never knows under what circumstances it is being used. This point does not seem to have an effect on language bindings.

6.13 Sigwait Functions and Unblocked Signals

Class: [ERR], [UND]

Reference: 3.3.8.2 [LINES: 213-215 (D9)]

Issue:

What happens if the signals in sigwait's set are not blocked?

Discussion:

Based on the usage pattern implied, this should not happen, and if it does, it should be treated as a user error.

6.14 Multiple Waiting Threads

Class: [UNS]

Reference: 3.3.8.2 [LINES: 219-220 (D9)]

Issue:

When multiple threads are waiting for the same signal, to which thread is the signal delivered?

Discussion:

This does not seem to be an important issue since applications cannot rely on such deterministic delivery. The delivery order inherently depends on the arrival pattern of the signals and their timing.

6.15 Real-Time and Non Real-Time Signals

Class: [UNS]

Reference: 3.3.8.2 [LINES: 231-233 (D9)]

Reference: 3.3.9.2 [LINES: 309-310 (D9)]

Issue:

"The selection order between realtime and nonrealtime signals, or between multiple pending nonrealtime signals, is unspecified."

Discussion:

Similar to 6.17, this does not seem to be a major issue.

6.16 Interrupting Sigwait Functions

Class: [ID]

Reference: 3.3.8.4 [LINES: 279-280 (D9)]

Issue:

Whether a signal can interrupt a sigwait function, and cause it to return with [EINTR].

Discussion:

It is implementation defined whether signals can interrupt (and resume) a thread waiting on a sigwait function. This presents a problem for a portable application since on the

one hand it must provide a handler for [EINTR], and on the other hand, it cannot rely on its ability to interrupt a sigwait blocking when the need arises.

6.17 Semantics of Sending a Signal to the Calling Process

Class: [IMP]

Reference: 3.3.9.2 [LINES: 305 (D9)]

Issue:

When *sigqueue()* sends a signal S to the calling process, which signal is delivered if other signals are already pending?

Discussion:

The text says: *"... either signo or at least one pending unblocked signal ..."*. It is not clear whether signal S is conceptually added to the pending set, followed by the normal selection (where lower numbered signals are delivered first), or whether there is a permission to unconditionally deliver S and ignore the other pending signals. It is also unclear what *"at least"* means here.

Another question concerns the treatment of multiple signals of the same number. There is no mention in the text of either a FIFO ordering or an implementation-defined order.

However, these problems are not considered serious since it is very unlikely that such pending and unblocked signals will in fact exist when *sigqueue()* is called.

6.18 Interactions Between *sleep()* and *alarm()*

Class: [IMP]

Reference: 3.4.2.2 [LINES: 659-662 (D8)]

Issue:

The relationship between *sleep()* and *alarm()*.

Discussion:

The *sleep()* and *alarm()* functions may both be implemented using the SIGALARM signal. It is unspecified what is the effect if both are used at the same time. From an application point of view, this means that the user is responsible for using one or the other. However, for an Ada RTS the problem is more serious. Such an RTS is likely to use the SIGALARM signal, but it does not have any simple and legal way to prevent a user from issuing *sleep()*, thus potentially interfering with the RTS' use of the only timing signal.

Chapter 7: Process Environment

This section describes portability issues related to Section 4 of P1003.4a.

7.1 Configurable System Limits

Class: [OPT]

Reference: 4.8.1 [LINES: 92-111 (D8)]

Issue:

All new configurable system limits.

Discussion:

Many values defined by POSIX APIs have configurable limits. This is basically the same fundamental issue discussed for 5.4 and 5.3.

Chapter 8: Files and Directories

This section describes portability issues related to Section 5 of P1003.4a.

8.1 Treatment of "dot" and "dot-dot" Directory Entries

Class: [UNS]

Reference: 5.1.2.2 [LINES: 23-24 (D8)]

Issue:

Whether *readdir_r()* returns directory entries for dot and dot-dot.

Discussion:

This may affect applications. However, portable applications can easily supply defensive-code to localize the effect.

Chapter 9: Asynchronous IO

This section describes portability issues related to Section 6.7 of P1003.4a.

9.1 The Priority of an Asynchronous IO Call

Class: [IHOL]

Reference: POSIX.4 6.17.1.1 [LINES: 393-414 (D8)]

Issue:

How is the asynchronous IO priority determined in a multi-threaded application?

Discussion:

This paragraph of POSIX.4 is not amended by P1003.4a. Since POSIX.4 talks only about processes and their priorities, many questions are left open and as a result it is not clear how this feature can be used at all in a multi-threaded application, let alone in a portable fashion. Some of the open questions are:

— Which one determines the priority of the asynchronous IO request: the thread's priority or the process'? A straight forward reading of the text suggests that it is the process; but then it is not very useful since all requests from a given process will have the same priority. Since, for execution scheduling, the process priority is always replaced by the thread priority, this rule is inconsistent with the rest of the treatment of process priorities. Note that as is explained in 13.8, the process priority does not really mean much when threads are supported.

— Which kind of priority is being used? The base or the active? Another aspect of this question is: Do changes to the priority of a thread/process affect the priority of pending asynchronous IO requests?

Chapter 10: C Language-Specific

This section describes portability issues related to Section 8 of P1003.4a.

10.1 Using Non-Thread-Safe Functions

Class: [ERR], [IMP]

Reference: 8

Issue:

The effect of using the non-thread-safe versions of the functions listed in this section in a multi-threaded application.

10.2 Unlocking a File Without Owning It

Class: [ERR], [UND]

Reference: 8.2.6.2 [LINES: 49-50 (D8)]

Issue:

The behavior when a thread unlocks a file without having first locked it.

10.3 Interrupted IO Functions on a Locked File

Class: [HOL]

Reference: 8.2.6.2 [LINES: 60-62 (D8)]

Issue:

Whether or not the IO functions mentioned here are required to call *funlockfile()* if they are interrupted (and return [EINTR]).

Discussion:

The text requires all POSIX.1 and C-language functions that refer to a locked file to use *flockfile()* and *funlockfile()* internally. It is not clear from the text whether the implementation of these functions is required to unlock the file if a given function call is interrupted.

10.4 Locked IO Operations by a Non-Owner

Class: [ERR], [IMP]

Reference: 8.2.7.2 [LINES: 115-119 (D8)]

Issue:

What happens if the functions listed in this paragraph are used without previously locking the corresponding file?

Discussion:

This is an error situation, but since the file in question may be used by another thread or even another process, documenting the effect is particularly important.

10.5 Non-Local Jumps

Class: [ERR], [UND]

Reference: 8.3.1.2 [LINES: 183-186 (D8)]

Issue:

The effect of calling *longjmp()* and *siglongjmp()* with a *buff* parameter that was saved by another thread.

10.6 Effect of *ctime_r* and *localtime_r* on *tzname*

Class: [MAY]

Reference: 8.3.6.2 [LINES: 273 (D8)]

Reference: 8.3.8.2 [LINES: 328 (D8)]

Issue:

Do the functions *ctime_r()* and *localtime_r()* set tzname?

Discussion:

Whether or not this happens seems to affect portable applications, however, the effect is local and the work-around is simple.

Chapter 11: Synchronization

This section describes portability issues related to Section 11 of P1003.4a.

11.1 Effect of Erroneous Behavior

Class: [ERR], [UND]

Issue:

What is the implementation response to erroneous behavior?

Discussion:

Note that the following are considered user "bugs". That is, we do not anticipate a legitimate need for exercising these cases.

- Using a non-shared semaphores outside its process 11.2.1.2 [LINES: 68-69 (D8)].
- Using a non-shared mutex outside the process 11.3.1.2 [LINES: 154-155 (D8)].
- Using a non-shared condition variable object outside the process 11.4.1.2 [LINES: 579-581 (D8)].
- Initializing an already-initialized mutex attributes object 11.3.1.2 [LINES: 132-133 (D8)].
- Initializing an already-initialized mutex 11.3.2.2 [LINES: 260-261 (D8)].
- Initializing an already-initialized condition variable attributes object 11.4.1.2 [LINES: 556-557 (D8)].
- Initializing an already-initialized condition variable object 11.4.2.2 [LINES: 643-644 (D8)].
- Using a destroyed mutex attributes object 11.3.1.2 [LINES: 141-142 (D8)].
- Using a destroyed condition variable attributes object 11.4.1.2 [LINES: 566-567 (D8)].
- Unlocking an unlocked mutex 11.3.3.2 [LINES: 473-474 (D8)].

11.2 Interrupting Semaphore Waits

Class: [ID]

Reference: 11.2.6.4 [LINES: 444-445 (D9)]

Issue:

Whether or not a signal can interrupt a semaphore wait function, and cause it to return [EINTR].

Discussion:

It is implementation defined whether signals can interrupt (and resume) a thread waiting on a semaphore. This presents a problem for portable applications since on the one hand they must provide a handler for [EINTR], and on the other hand, they cannot use a signal to wake up a task blocked on a semaphore lock operation.

11.3 Destroying a Synchronization Object With Blocked Threads

Class: [ERR], [UND]

Reference: 11.2.2.2 [LINES: 73-74 (D8)]

Reference: 11.4.2.2 [LINES: 651-652 (D8)]

Issue:

The effect of destroying a semaphore with blocked threads, or of destroying a condition variable object that has threads blocked on it.

Discussion:

In most cases, this is an error situation. However, for applications that wish to provide error-recovery (and an Ada RTS is one such application with respect to its own resources), this may be an important issue. It is often necessary to destroy a synchronization object asynchronously; even if there was a reliable way to get the number of threads blocked on that object, it would be impractical to either wait for them to leave the queue or to kill them. As a result, the entire error-recovery approach becomes non-portable.

This issue also affects language bindings which attempt to provide a safer interface (e.g. Ada). This is because such a binding cannot portably rely on the appropriate support from the P1003.4a implementation.

11.4 Resources Used By Held Mutexes

Class: [IMP]

Reference: 11.3

Issue:

Does a thread waiting on a mutex hold the processor, or tie up the memory bus?. What is the effect of a thread waiting on a mutex, on the execution of other threads?

Discussion:

We believe that this issue has a special importance; in particular for real-time applications. There are different implementation approaches to mutexes because they have

to trade-off between speed and bounded execution time. This is particularly true for multi-processor systems, since there are so many different approaches to solving the inter-connection problem.

The two most extreme alternatives are probably the spin-lock and the queuing models. This implementation choice is relevant to the timing of a real-time application since it determines: (1) Whether a thread keeps its processor busy while waiting for a mutex, (2) whether priority queuing is possible, and (3) whether implementation-induced deadlocks are possible (in the extreme case where the number of waiting threads is larger than the number of available processors).

11.5 Implementation-Defined Attributes of Attributes Objects

Class: [ID]

Reference: 11.3.1.2 [LINES: 129-131 (D8)] and [LINES: 169-170 (D8)]

Reference: 11.4.1.2 [LINES: 553-555 (D8)] and [LINES: 595-596 (D8)]

Issue:

The members, default values, and access functions of implementation-defined attributes of mutex and condition variable attributes objects.

Discussion:

Portable applications do not seem to need more implementation-defined attributes. Therefore the effect of this point is limited to applications that explicitly use extensions.

Language bindings must provide a standard way to introduce such implementation-provided extensions.

11.6 Return Values from Destroy Operations

Class: [MAY]

Reference: 11.3.1.2 [LINES: 138-140 (D8)]

Reference: 11.3.2.2 [LINES: 263-265 (D8)]

Reference: 11.4.1.2 [LINES: 563-565 (D8)]

Reference: 11.4.2.2 [LINES: 646-648 (D8)]

Issue:

Whether the *pthread_mutexattr_destroy()*, *pthread_condattr_destroy()*, *pthread_mutex_destroy()*, and *pthread_cond_destroy()* operations set an invalid value in *attr*, *attr*, *mutex*, and *cond*, respectively, upon return.

Discussion:

Such behavior is useful and increases the safety of an application. However, if needed, it can be done redundantly by a user code in a fairly localized fashion. Language bindings may also choose to provide this safer approach. The implementation of the language binding can provide the same semantics; it does not need to rely on the underlying P1003.4a implementation.

11.7 Using Destroyed Synchronization Objects

Class: [ERR], [UND]

Reference: 11.3.2.2 [LINES: 266-268 (D8)]

Reference: 11.4.2.2 [LINES: 649-650 (D8)]

Issue:

The effect of using a destroyed mutex or a destroyed condition variable object, and the effect of destroying a locked mutex.

Discussion:

In most cases, this usage will be considered a bug. However, such a functionality may be needed for error recovery when the thread that has declared the object cannot be relied upon to handle the situation in a timely fashion, (it may be dead). Also, in multi-threaded applications, a synchronization object may be destroyed when other threads are still active. This is essentially a timing issue. To solve it without this functionality will require an additional synchronization layer. Providing such a mechanism to prevent this (or to communicate the object's destruction to all threads) may defeat the whole purpose of the P1003.4a synchronization objects to begin with and may itself require a better-defined behavior for this case.

11.8 Uncoordinated Mutex Lock/Unlock

Class: [ERR], [UND]

Reference: 11.3.3.2 [LINES: 465-466 (D8)] and [LINES: 471-472 (D8)]

Reference: 11.3.3.4 [LINES: 508-513 (D8)]

Issue:

The effect of re-locking a mutex by its owner or of unlocking a mutex by a thread other than the owner.

Discussion:

These are usually error situations. However, the need to lock a mutex from a signal handler is needed in some design approaches for the implementation of an Ada RTS. In such cases, it is not always known whether the caller thread owns the mutex. This capability is particularly necessary since P1003.4a does not provide a way to query the identity of a mutex owner. It is true however, that such a capability by itself would not

be enough; calling *pthread_mutex_lock()* would have to be async-signal-safe as well (see 6.5).

Unlocking a mutex by a thread other than its owner, is a capability often needed for applications that do their own error recovery (such as an Ada RTS). The owner thread may already be gone or it may be in a state which cannot be relied upon to unlock the mutex in a timely fashion. Leaving the mutex alone and allocating another mutex for the same resource is not always feasible. First, there may be other active threads blocked on the thread, and second, it is not trivial to asynchronously communicate the identity of the new mutex to all interested threads.

A related question is whether these errors are detected by the implementation. Although this is a normal case of *"if detected"*, it is more significant because of the potential benefits of the usages mentioned here.

11.9 Mutexes held by Terminated Threads

Class: [ERR], [IMP]

Reference: 11.3.3

Issue:

What is the result of terminating a thread which is the owner a mutex?

Discussion:

P1003.4a does not require implementations to do anything special in this case such as returning an error code or implicitly unlocking the mutex. However, implementations may attempt to provide a protection or recovery mechanisms. Whether or not this is done, is of course relevant to the application, but in general we consider this case (letting a thread terminate while owning a mutex) to be a user error.

11.10 The Timing of a Ceiling Check

Class: [IMP]

Reference: 11.3.3.4 [LINES: 497-499 (D8)]

Issue:

When is the check for ceiling violation performed?

Discussion:

A ceiling violation is when a thread attempts to lock a mutex while the thread's priority is higher than the mutex ceiling priority. It is not clear when this check has to be performed. There are two possibilities: Either immediately, when the *pthread_mutex_lock()* or *pthread_mutex_trylock()* are called, or when the calling thread has reached the head of the queue, and is about to become the owner of the mutex. The difference is visible when the priority ceiling of a mutex is dynamically changed, or when the thread's priority is

changed. The effect is also visible, in general, during debugging and when doing error recovery. This does not seem to be an issue for an Ada RTS implementation or for an Ada binding since it is unlikely that the Ada binding will export the ability to do priority ceiling changes to the application.

11.11 Spurious Wake-Ups from Condition Signaling

Class: [IMP]

Reference: 11.4.3.2 [LINES: 705 (D8)]

Issue:

Can *pthread_cond_signal()* wake up more than one thread? (i.e. are spurious wake-ups allowed?)

Discussion:

The text in 11.4.3.2 permits spurious wake-ups: *"The pthread_cond_signal() call unblocks at least one of the threads that are blocked on the specified condition variable ... "* This is a major portability concern which is acknowledged by the rationale of P1003.4a. The rationale suggests a coding style to safe-guard against such spurious wake-ups: *"It is thus recommended that a condition wait be enclosed in the equivalent of a "while loop" that checks the predicate."* While feasible, this model incurs more overhead for the "normal" cases. In addition, this may be of concern for applications which try to minimize the number of context-switches (or to have a deterministic number of them). It seems a poor trade-off for the standard to require users to protect themselves against deficiencies in the P1003.4a implementation.

11.12 State Transitions of a Signaled Thread

Class: [IMP]

Reference: 11.4.3.2 [LINES: 710-718 (D8)]

Issue:

Does a thread become *ready* after a signal wakes it up and before it contends for the corresponding mutex?

Discussion:

One sentence in this paragraph:

> *"The thread(s) that are unblocked shall contend for the mutex according to the scheduling policy(if applicable), and as if each had called pthread_mutex_lock(). "*

can be interpreted as requiring that the thread becomes formally *ready* before it starts to contend for the mutex. The first clue is the term "unblocked" which implies that the

thread joins a process-list in the ready queue. Secondly, the fact that the behavior is *as if* a normal mutex lock was initiated, implies that the thread was *ready* before.

We do not believe that this was the intention; on the contrary, the idea is to allow the implementation to move the thread directly from the condition variable queue to the mutex queue. However, the current wording clearly suggests this is not allowed.

The significance of this issue lies in the possibility of having side-effects as a result of going through the *ready* state. Specifically, there is no requirement to maintain the original FIFO order (from the condition variable queue) on the mutex queue when the following conditions are met: SCHED_FIFO is used on a multi-processor system, and the implementation uses the permission to unblock more than one thread when a signal arrives. This is due to the fact that the threads are considered *ready* and therefore they may execute. On a multi-processor, there is no notion of relative speed or order of execution, and hence these threads may queue on the mutex in a different order than the one which existed for the condition variable queue.

11.13 Errors When Mutexes are Used Implicitly

Class: [ERR], [UND]

Reference: 11.4.4.2

Issue:

Error situations associated with the implicit use of mutexes.

Discussion:

Condition variable wait operations use a mutex as a parameter. Implicit locking and unlocking is performed by the *pthread_cond_wait()* and *pthread_cond_timedwait()* operations. Therefore, error conditions, similar to those exist for the other mutex operations, may arise when:

— The specified mutex is a destroyed mutex, or the mutex is destroyed between the implicit unlock and the implicit re-lock.

— The specified mutex is in an unlocked state at the time of the call.

— The specified mutex is locked by another thread at the time of the call.

— Either the priority ceiling of the mutex or the priority of the blocked thread has been changed so that when the thread is awakened and implicitly re-locks the mutex, a ceiling violation results.

The above points are discussed in more detail in the sections containing the corresponding explicit operations.

11.14 Multiple Threads Waiting on One Condition Variable With Different Mutexes

Class: [ERR], [UND]

Reference: 11.4.4.2 [LINES: 854-858 (D8)]

Issue:

The effect of using *pthread_cond_wait()* or *pthread_cond_timedwait()* with the "wrong" mutex.

Discussion:

Here, "wrong" means:

> *" The effect of using more than one mutex for concurrent pthread_-cond_wait() or pthread_cond_timedwait() operations on the same condition variable is undefined — that is, a conditions variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding ends when the wait returns. "*

This bug may happen quite frequently.

11.15 Rules for Asynchronous Cancelability

Class: [HOL]

Reference: 11.4.4.2 [LINES: 859-874 (D8)]

Issue:

Missing semantics for the PTHREAD_CANCEL_ASYNCHRONOUS case.

Discussion:

This paragraph talks only about the PTHREAD_CANCEL_DEFERRED case and not about the general PTHREAD_CANCEL_ENABLE case (i.e. the case of PTHREAD_-CANCEL_ASYNCHRONOUS is missing). This is probably a simple omission and we assume that the rules are intended to be identical. This is also based on the text in 18.1.2 [LINES: 239-242 (D8)] which supports this interpretation.

11.16 Signal Delivery and Condition Waits

Class: [MAY]

Reference: 11.4.4.2 [LINES: 884-887 (D8)]

Issue:

Whether an interrupted call to *pthread_cond_wait()* or *pthread_cond_timedwait()* resumes waiting after the signal handler returns, or the call returns spuriously.

Discussion:

Naturally, this is of major concern to applications since it affects the fundamental use of condition variables. The practical problem is that many implementations are likely to

provide reliable wakeup on signal delivery, and applications are likely to take advantage of it and rely on this behavior, thus becoming non-portable. The work-around should be the same as the one that protects against spurious wake-ups, so the effect can be made local.

Chapter 12: Memory Mapping

This section describes portability issues related to Section 12.2 of P1003.4a.

12.1 *msync()* in Multi-Threaded Applications

Class: [IHOL]

Reference: 12.2.4.2 POSIX.4

Issue:

What is the relationship between the execution of the *msync()* operations and the other threads that execute at the same time?

Discussion:

"The msync() function writes all modified data to permanent storage locations, ... ". This part of POSIX.4 is not amended by P1003.4a, therefore the context is still a single-threaded application. It is natural to assume in such an environment that nothing else will execute in the process while *msync()* runs. However, the same assumption cannot be made when multiple threads execute concurrently. It is neither practical, nor always possible to stop all other threads while the *msync()* executes. Therefore, it makes sense to wonder whose responsibility it is (application vs. implementation) to ensure that no conflicting memory accesses occur.

Note that similar issue arises with file synchronization operations, and therefore, the standard defines them as atomic. No such requirement is specified here (nor can it be easily implemented).

Chapter 13: Execution Scheduling

This section describes portability issues related to Section 13 of P1003.4a.

13.1 Introduction

The issues described below have a major effect on the design of any portable, real-time system. Of particular concern is the issue of an Ada 9X RTS supporting the Real-Time Annex. The scheduling rules in POSIX and the Ada 9X's Real-Time Annex are quite similar and were developed with an emphasis on coordination. However, while so many issues in POSIX are left under-specified, the Ada 9X standard is very specific in its scheduling rules. As a result, an Ada RTS will have a hard time relying on the underlying P1003.4a implementation to provide the appropriate semantics and will have to either ignore the P1003.4a implementation and provide its own support or, worse, work around holes in the semantic description of POSIX. This is true even if we believe that most implementations of P1003.4a will eventually do the "right thing".

13.2 The Effect of Scheduling Policy on Resource Management

Class: [IMP]

Reference: 13

Issue:

What other processing resources are controlled by the scheduling policies? Are they also included in the contention scopes?

Discussion:

POSIX requires only CPU scheduling to be done according to the rules described in section 13. It recommends applying them to the scheduling of other system resources as well. But there is no requirement to do so. On the face of it, this seems to be an important issue. However, it does not seem likely, in the near future, that resource-based scheduling will become a standard practice, and thus portable. Therefore it seems that applications that use such vendor-specific capabilities are unlikely to be portable in any case.

13.3 Base vs. Active Priorities and Queuing

Class: [HOL]

Reference: 13

Issue:

Which priority is used for the queuing operations? The base priority or the active one? (See 2.2 for a detailed discussion on the subject of base vs. active priorities.) If the active priority is used for queuing, do changes to it affect the location of the thread/process in a queue?

Discussion:

These questions are not adequately addressed by either POSIX.4 or P1003.4a. The discussion about mutexes suggests that the active priority is used, but this discussion is not conclusive and does not cover other areas such as semaphores, message queues, etc.

The answer to this question is important since in POSIX, threads can execute for a long time with an active priority that is different from the base priority. If no clear semantics exist, real-time applications, which must ensure schedulability of their threads, cannot be developed in a portable way. In fact, since the actual behavior is not categorized as "implementation defined", it is not clear that even non-portable applications can be developed since it may be impossible for a user to determine the actual behavior supported by the implementation. (Since no explicit rules are given, implementors may miss this issue all together, and potentially use arbitrary and therefore inconsistent solutions.)

13.4 The Interaction Between Inheritance and Dynamic Priority Changes

Class: [HOL]

Reference: 13

Issue:

What are the interactions between dynamic priority changes (i.e. via *pthread_setschedparam()*, 13.5.2.2) and other rules which implicitly change the active priority?

Discussion:

The general issue (which is critical to real-time applications and the ability to use P1003.4a as the platform for Ada RTSs) is the interaction between user-initiated priority changes (i.e. to the base priority) and the cases where the active priority of a thread is changed as a result of other operation (such as the locking of a mutex by a high priority thread). A particular problem to POSIX is that since the distinction between the two kinds of priorities are not formalized, the answers to these questions have to be given for each case of queuing and/or inheritance separately. The following are common questions that have to be addressed where the two models of priority changes co-exist:

— When *pthread_setschedparam()* is called on a thread that has its active priority different than its base one, does the new base priority override the current active priority if it is higher? When the source of inheritance ceases, to what value does the base priority revert?

— Which priority (base or active) affects the thread's position in queues? Do changes to that priority (i.e. base or active) while the thread is on a queue, cause reordering of the queue?

— Which priority is inherited by other threads in the various cases?

— Do priority changes of a donor thread affect the entire inheritance tree?

See also 13.17 for a related discussion on ceiling priorities.

13.5 Thread's Preemption and Thread-List

Class: [HOL]

Reference: 13

Issue:

When a thread is preempted, does it join the thread-list of its active or its base priority?

Discussion:

A running thread is not on any thread-list. When it is preempted, it re-joins the thread-list corresponding to its priority, but it is not clear from the text whether this is done based on the base or active priority of the thread. This is probably just an omission. For proper behavior, it is clear that the active priority should be used in this case.

13.6 Ordering of Queues

Class: [HOL]

Reference: 13

Issue:

Ordering of various queues.

Discussion:

Where global system resources such as semaphores, message queues, and others are concerned, two questions arise:

— What is the queuing order when the priority scheduling is not supported? and more importantly,

— What is the relationship between the processes' and the threads' priorities when threads from different processes are queued on the same resource (including the case of single-threaded processes).

The answers to these questions are important for any application. Even if priority scheduling is not used, applications need to know whether the queuing order is fair, deterministic, or follows some other rules.

The second point is critical for any application that requires bounded time blocking for scheduling analysis. This question is particularly important, since (as discussed in 13.8) the meaning of process priority is not properly defined.

13.7 Process vs. Thread Priority

Class: [IHOL]

Reference: 13.2 [LINES: 468-469 (D9)]

Issue:

Missing or conflicting semantics when integrating P1003.4a into POSIX.4 with respect to the meaning of priorities.

Discussion:

P1003.4a amends POSIX.4 by replacing all occurrences of "process" in 13.2 with "thread". That is an appropriate amendment in general, but it is not complete. To begin with, several function names appear in the description of 13.2; these functions are important to the semantics of the scheduling policies. However, these function names are of the "process" versions, not the "thread" ones. For example: *sched_setscheduler()* in [LINES: 32 (D8)] and [LINES: 60 (D8)], and *sched_setparam()* in [LINES: 32 (D8)] and [LINES: 65 (D8)]. It is not clear whether or not a substitution of function names is intended here as well. Furthermore, in other parts of P1003.4a (such as in 13.3.1.2), the term "process list" is still being used (instead of "thread-list"), making the semantics ambiguous. Finally, it is unclear (with this amendment) what is the meaning of process' priority (as defined in POSIX.4 13.1), since the rest of the discussion is in terms of threads' priorities.

13.8 The Meaning of Process Priorities

Class: [UNS]

Reference: 13.3.1.2 [LINES: 477-480 (D9)]

Reference: 13.3.3.2 [LINES: 486-489 (D9)]

Reference: 13.4.2 [LINES: 213-223 (D8)]

Issue:

What is the meaning of the process' priority (under both contention scopes) where threads are supported (but not necessarily used)?

Discussion:

P1003.4a amends POSIX.4 in a way that makes the process' priority concept useless. Either it is completely irrelevant (as is the case with PTHREAD_SCOPE_SYSTEM where threads compete based on their own priority), or it is "unspecified". The latter is because the discussion about threads scheduling under PTHREAD_SCOPE_PROCESS

(see 13.4.2 [LINES: 221-223 (D8)]) reads: *"It is unspecified how such threads are scheduled relative to threads in other processes or threads with* PTHREAD_SCOPE_SYSTEM *scheduling contention scope."* This means that using process scheduling scope does not really guarantee any scheduling behavior on any system that is multi-process.

Note that the question about the meaning of the process priority is relevant even when threads are not being used, since P1003.4a defines each process to have at least one thread and it is that thread's priority that is used for scheduling.

Furthermore, since the text uses words such as *"unspecified"* and *"may be dependent"*, the implementation is not even required to document its chosen behavior, nor is it required to behave the same each time a scheduling decision is made.[1] It is not entirely clear what this means and which of these three apparently inconsistent statements takes precedence.

In light of these problems, and since an application cannot rely on any specific contention scope to actually be supported (see 13.12), we believe that it will be impossible to write portable real-time applications using the POSIX facilities.

13.9 Yielding in the Absence of Priority Scheduling

Class: [HOL]

Reference: 13.3.5.2 [LINES: 498-503 (D9)]

Issue:

What are the rules for selecting a new thread after a *pthread_yield()* when POSIX_THREADS_PRIORITY_SCHEDULING is not defined?

Discussion:

Clearly, this is not an issue for real-time applications (since it arises only when no priority scheduling is used). However, other applications, such as servers, might want to employ some sort of rudimentary fair service order. The function *pthread_yield()* could be a likely candidate to achieve this, but as the revised text of P1003.4a/D9 stands, the semantics of this function are described in terms of "thread lists". This does not make sense without priority scheduling. Furthermore, it is unclear whether the intent is to allow this function when POSIX_THREADS but not POSIX_THREADS_PRIORITY_SCHEDULING is defined.

As a result, portable applications cannot use this function safely under this set of options.

13.10 Mixing Scheduling Policies

Class: [ID]

Reference: 13.4.1 [LINES: 180-181 (D8)]

[1] However, in 13.4.2 [LINES: 213-217 (D8)], the mapping of the scheduling attributes into system attribute space is implementation-defined.

Reference: 13.4.2 [LINES: 221-223 (D8)]

Issue:

The effect of having threads with different policies or contention scopes in a given process.

Discussion:

This seems to be a minor issue. It is not likely that applications will actually benefit from such a mixture and, if they do, it does not seem that portability is meaningful in this case.

13.11 Default Scheduling Attributes

Class: [ID]

Reference: 13.4.1 [LINES: 191-193 (D8)]

Issue:

The default values of all scheduling attributes, in particular the default contention scope of a thread.

Discussion:

This may be interesting to the user, but a portable application can do without relying on the defaults (except for a short time immediately after a process is created). Implementation-defined attributes are inherently non-portable.

13.12 Support of Contention Scopes

Class: [OPT]

Reference: 13.4.2 [LINES: 224-225 (D8)]

Issue:

Which contention scopes are actually supported?

Discussion:

P1003.4a allows an implementation to support PTHREAD_SCOPE_SYSTEM, or PTHREAD_SCOPE_PROCESS, or both. Portable applications cannot rely on any one of these to be supported in a given system. As is discussed in 13.8, the effective contention scope is essential to the meaning of process priorities. The answer to this question is crucial to the timing of the application, and therefore this non-portability is significant.

13.13 How are Allocation Domains Specified?

Class: [ID]

Reference: 13.4.3

Reference: 13.4.4

Issue:

The specification of allocation domains in a given system.

Discussion:

What are the allocation domains in a system? Are they per-thread, per-process, or per-system? Are they static or dynamic? If user control is available, how this is done?

Note that there is no way to query the size of the allocation domain from inside the application, and even on a single processor system the size is not guaranteed to be one (see 4.4).

This information is important for all systems since the scheduling policy in P1003.4a is defined only for allocation domains of one.

13.14 Semantics of Allocation Domains Larger Than One

Class: [ID]

Reference: 13.4.3

Reference: 13.4.4

Issue:

Semantics of allocation domains larger than one.

Discussion:

For allocation domains larger than one: What are the scheduling policies? For SCHED_-FIFO and SCHED_RR what is the interpretation when the allocation domain is larger than one?

For applications that rely on such allocation domains, this is important information. However, this aspect of scheduling is not likely to become standardized any time soon, so the lack of portability is not of major concern.

13.15 Implementation-Defined Scheduling Policies

Class: [ID]

Reference: 13.4.4

Issue:

Other scheduling policies, their attributes, and their interactions with all other available policies.

Discussion:

This is useful information about a given implementation, but relying on such information is inherently non-portable.

13.16 Semantics of SCHED_OTHER

Class: [ID]

Reference: 13.5.2.2 [LINES: 426-427 (D8)] and [LINES: 439-440 (D8)]

Issue:

What is the meaning of the SCHED_OTHER policy with respect to attributes, interactions with other policies, and the use of contention scopes? Is it equivalent to SCHED_FIFO or SCHED_RR?

Discussion:

Some applications may use the SCHED_OTHER as a default policy when real-time scheduling is not needed. In those cases, it does not seem likely that the specifics of this policy will become relevant to the application (presuming the policy does sensible things).

13.17 PRIO_PROTECT and Priorities

Class: [IMP]

Reference: 13.6.1.2 [LINES: 594-624 (D8)]

Issue:

Which priority is used for ceiling checks and for inheritance under PTHREAD_PRIO_PROTECT?

Discussion:

The rule that says that a donor's active priority is inherited by other threads (as opposed to the base priority) is specified only for the PTHREAD_PRIO_INHERIT case, and not for PTHREAD_PRIO_PROTECT. Which priority is used in the ceiling-check? Is the priority value inherited from a mutex ceiling used as the thread's own active priority when that thread is later blocked?

13.18 Changing Ceiling Priorities

Class: [IMP]

Reference: 13.6.2.2

Reference: 13.6.2.2 [LINES: 576-577 (D9)]

Issue:

The semantics of *pthread_mutex_setprioceiling()*.

Discussion:

When a thread issues a *pthread_mutex_setprioceiling()* call and the mutex is locked, the calling thread will be blocked. Does it join the queue based on its priority or at its (the queue's) head? Furthermore, if the latter is true, then after the ceiling is changed, the associated mutex queue may be nonempty; is the ceiling check (with the new value) now being performed for all the threads on the queue?

In addition, the semantics of this operation include the locking of the mutex, but the effect of calling it while holding the mutex are not specified (no [EDEADLK] is defined). On the other hand, it may be quite useful to allow the calling thread to be the owner of the mutex whose priority is being changed (see below), but this is apparently disallowed.

Another problem was introduced by draft 9 of P1003.4a, which says *"The process of locking the mutex need not adhere to the priority protect protocol."* In draft 8, "shall not" was specified where now "need not" appears. This change has a fairly substantial and negative effect on users. Portable applications cannot tell whether the thread that changes the mutex ceiling priority has to obey the rules of the priority protect protocol. The problem with this is that the thread that affects ceiling changes is usually some sort of a manager (or a mode-change) thread that must run in a high priority for unrelated reasons. In order to be safe, this thread will have to lower its priority (to avoid probable ceiling violations) before it calls *pthread_mutex_setprioceiling()*. This will completely change the timing behavior of the application.

This feature has limited applicability; it is useful only for applications that employ sophisticated mode-changes. Even in these situations, it is not yet clear how well such a ceiling-change will be performed in practice. For those applications that find this feature useful, the answers to the above questions are very important since they dictate the high-level design of the mode-change; namely whether the mutex queue has to become empty (on its own, or by an explicit program action) before its ceiling priority can be changed or whether the change can be done "asynchronously" while the mutex is still in active use.

The inescapable conclusion of this analysis is that this feature cannot be used by portable applications. The benefit of its inclusion in a standard is therefore questionable.

Chapter 14: Clocks and Timers

This section describes portability issues related to Section 14 of P1003.4a.

14.1 Semantics of Timer-Generated Signals

Class: [IHOL]

Reference: 14.2.2.2 POSIX.4 [LINES: 157-163 (D14)] and [LINES: 168-174 (D14)]

Issue:

Can timers use the extended signal notification mechanisms?

Discussion:

P1003.4a adds many capabilities to the way signals can be delivered (such as invoking a "notification-thread"). P1003.4a does not amend this clause and it is not clear that the text can be interpreted in such a way as to allow these additional capabilities. (For example, notification threads and queuing signals are not mentioned.) At the minimum, a friendly interpretation is needed.

Chapter 15: Message Passing

This section describes portability issues related to Section 15 of P1003.4a.

15.1 Closing a Message Queue

Class: [IHOL]

Reference: 15.2.2.2 POSIX.4

Issue:

The effect of closing a message queue while other threads still have access to it.

Discussion:

This paragraph is not amended by P1003.4a, and therefore the issue of inter-process concurrency is not discussed. The assumption is that there is only one thread of control within the process that calls *mq_close()*. With multiple threads, one thread may close a message queue while other threads of the same process may be blocked on it (in fact, other threads may be in the middle of reading or writing to/from this queue).

This should probably be classified as [ERR], but P1003.4a is silent about it.

15.2 Which Priority is Used for Arbitration on Message Queues?

Class: [IMP]

Reference: 15.2.4.2 [LINES: 8 (D8)]

Issue:

Is the base or the active priority used to arbitrate among threads attempting to insert a message into a queue?

Discussion:

This is a very important question for real-time applications. A thread may execute a long time with an active priority different from its base priority and during this time it may send messages. Note that the issue of the thread's priority is relevant only when more than one thread is waiting to insert a message to the queue; it does not affect the order of messages while they are already in the queue (a separate parameter is provided for this purpose). The only issue that affects the order of messages in the queue is the requirement to queue equal priority messages in FIFO order. The result of the arbitration will affect the FIFO order. Admittedly, this is an extremely subtle issue.

15.3 Order of Message Queue Operations

Class: [UNS]

Reference: 15.2.4.2 [LINES: 9-10 (D8)]

Reference: 15.2.5.2, [LINES: 605-606 (D9)]

Issue:

What are the rules for selecting a waiting thread (to send or receive a message) when the priority scheduling option is not supported?

Discussion:

This is not a real-time issue, but it is important for any portable application that must ensure some sort of fair or deterministic service. The fact that implementations are not required to document the order they use, aggravates the problem.

Chapter 16: Thread Management

This section describes portability issues related to Section 16 of P1003.4a.

16.1 Reclamation of Threads' Resources

Class: [IMP]

Reference: 16

Issue:

When, if ever, are the system resources associated with threads reclaimed after *pthread_exit()* or *pthread_join()*?

Discussion:

There is no requirement that all of the resources that are allocated by the system for a thread be reclaimed (and thus, be available for the use of other threads) immediately when these functions return. This is important for applications that use many threads in a resource-tight system.

16.2 Uniqueness of Thread-Id Values

Class: [UNS]

Reference: 16.1 [LINES: 8-12 (D8)]

Issue:

Are the values used for thread-id (*pthread_t*) unique in a system? What is the effect of using a value of *pthread_t* from another process?

Discussion:

The answers to these question should not affect well-behaved applications. It may potentially affect the behavior of the implementation when a user error occurs.

16.3 Whether Thread-Ids are Recycled

Class: [IMP]

Reference: 16.1 [LINES: 12-16 (D8)]

Issue:

Are the values of *pthread_t* recycled?

Discussion:

This issue will affect long-running, large applications. The issue is whether or not a newly-created thread may be assigned a thread-id that has previously been assigned to a thread which has long been gone. This thread-id value may still be stored in one of the application data structures and cause confusion.

16.4 Implementation-Defined Thread Creation Attributes

Class: [ID]

Reference: 16.1.1.2 [LINES: 64-65 (D8)]

Issue:

What are the implementation-defined thread creation attributes? What are the default values for all thread creation attributes?

Discussion:

In general, this should not affect the portability of applications. For standard attributes, the dependence on the default values can be avoided by a local solution. The use of additional implementation-provided attributes is inherently non-portable.

16.5 Return Values From the Destroy Operation

Class: [MAY]

Reference: 16.1.1.2 [LINES: 71-72 (D8)]

Issue:

Whether the *pthread_attr_destroy()* operation sets an invalid value in *attr* upon return.

Discussion:

Such behavior is useful and increases the safety of an application. However, if needed, it can be done redundantly by a user code in a fairly localized fashion. Language bindings may also choose to provide this safer approach. The implementation of the language binding can provide the same semantics; it does not need to rely on the underlying P1003.4a implementation.

16.6 Using a Destroyed Thread Creation Attributes

Class: [ERR], [UNS]

Reference: 16.1.1.2 [LINES: 72-73 (D8)]

Issue:

The effect of using a thread creation attribute object after it has been destroyed.

16.7 Stack Size Padding

Class: [IMP]

Reference: 16.1.1.2 [LINES: 74-78 (D8)]

Reference: 3.3.1.3

Issue:

The amount of padding added to the stack size; (e.g. for headroom for signal handlers, nested signal handlers, etc.).

Discussion:

Applications that set the stack-size option presumably have a reason to do so: Either the default is too small, in which case the thread is likely to overflow its stack space, or the default is too high, in which case storage may be wasted. The latter is particularly relevant to applications which have many simultaneously active threads, but with limited memory availability. The default value may not be known, so the application usually has to set the stack size value to ensure correct execution.

Most Ada RTSs support user control over the stack size, so knowing the amount of padding is important there as well.

Finally, signals may arrive in such a way that their handlers are invoked in a nested fashion. The stack must be large enough to accommodate the maximum nesting level. This maximum is probably indeterminable, but a practical value can be established. (If the nesting level exceeds this maximum, the application will fail.)

It is not clear from the text, (and there are no corresponding documentation requirements), whether stack frames for signal handlers are taken from the allocated size of the stack and whether part of the stack can be used by the signal handlers of the P1003.4a implementation itself. If such was the case, not every byte on the stack is available to the thread. Alternatively, some slack may be added by the implementation. In this case, the question is what is the size of this slack? To summarize, is it the user's responsibility to allocate a stack that is large enough to accommodate all nested handlers?

16.8 The Response to a Stack Overflow

Class: [IMP]

Reference: 16

Issue:

What is the implementation response to a stack overflow?

Discussion:

While the entire issue of stack handling is fairly system-specific, most systems have stacks, and for applications with high-reliability requirements, it is important to know the answer to this question so that proper recovery can be implemented. Note that is is implementation-dependent whether the check for a stack overflow is done at all, and if a violation is detected, what signal is generated.

16.9 Use of Thread Ids of Detached Threads

Class: [HOL]

Reference: 16.1.1.2 [LINES: 613-620 (D9)]

Reference: 16.1.3.4 [LINES: 339-340 (D8)]

Reference: 16.1.4.2 [LINES: 664-668 (D9)]

Issue:

What is the effect of using the thread-id of a detached thread in a *pthread_join()* call? What is the effect of multiple simultaneous calls on *pthread_join()* for the same target thread?

Discussion:

It is unclear whether this behavior is unspecified, undefined, or required to return an error-code. With the re-introduction of *pthread_detach()* in D9, the effect of this operation is *not* described in terms of the *detachstate* attribute, but instead, the text repeats the semantics of that attribute (see 16.1.4.2 [LINES: 664-667 (D9)]). The question is whether a thread is considered "detached" after *pthread_detach()* is called on it.

Specifically: 16.1.1.2 [LINES: 613-614 (D9)] says: *"If a thread is detached, its thread-id is invalid for use as an argument in a call to pthread_detach() or pthread_join()."* The meaning of "invalid" is usually that an error code is returned, as is in fact specified in 16.1.3.4 [LINES: 339-340 (D8)]. However, 16.1.1.2 [LINES: 619-620 (D9)] says: *"If the thread is created detached, then use of the ID of the newly created thread to the pthread_detach() or pthread_join() function is undefined."* Furthermore, 16.1.4.2 [LINES: 667-668 (D9)] claims that: *"The effect of multiple pthread_detach() calls on the same target thread is unspecified."* However, two error codes are nevertheless specified for this case ([EINVAL] and [ESRCH]).

Clearly, more work is needed here in order to clarify the rules. However, it is also clear that the intent is to leave some of the rules with respect to errors as not fully specified. Unfortunately, this means that having multiple threads "joining" one target thread and having multiple threads do "detach" on one target thread is considered by the standard to be an illegitimate paradigm. Such a paradigm may be quite useful for multiple threads that all wish to wait for another thread's termination, or when threads want to express their disinterest in another thread, *independently*. Instead, some form of manager thread will have to be implemented to perform this function on behalf of the other threads, and these other threads will somehow have to communicate with this manager thread.

16.10 The Return Value From *pthread_create()*

Class: [ERR], [UND]

Reference: 16.1.2.2 [LINES: 237-238 (D8)]

Issue:

The value returned in *thread* when *pthread_create()* fails.

16.11 Count of Terminated Threads

Class: [UNS]

Reference: 16.1.3.2 [LINES: 327-328 (D8)]

Issue:

Whether terminated but unjoined threads count against {_POSIX_THREAD_THREADS_MAX}.

Discussion:

This point is important for large or long-running applications which may reach the limit on concurrent active threads. Since the behavior is "unspecified", there is no documentation that the user can consult in order to find out the answer before coding starts.

16.12 Erroneous Thread Exit

Class: [ERR], [UND]

Reference: 16.1.4.2 [LINES: 391-393 (D8)]

Issue:

The effect of calling *pthread_exit()* from cleanup handlers or thread-specific data destructors.

16.13 Order of Calling Thread-Specific Data Destructors

Class: [UNS]

Reference: 16.1.4.2 [LINES: 383-384 (D8)]

Reference: 17.1.1.2 [LINES: 96-98 (D9)]

Issue:

In what order are thread-specific data destructors called?

Discussion:

This should not be a portability concern because it is not considered a safe and "correct" practice to rely on such an order when dealing with finalization.

16.14 Choice of Exit Status

Class: [HOL]

Reference: 16.1.4.2 [LINES: 394-396 (D8)]

Reference: 16.1.2.2 [LINES: 231-233 (D8)]

Issue:

Which exit status is used when the main thread terminates?

Discussion:

The two sentences referenced here seem to conflict. 16.1.2.2 [LINES: 231-233 (D8)] says that when *main()* returns, its return value is used for the exit status. However 16.1.4.2 [LINES: 394-396 (D8)] says that "0" returned *"... after the last thread has been terminated."*

The source of the problem is that the term *thread termination* is not formally defined. The name of 16.1.4 is "Thread Termination", but it mostly discusses *pthread_exit()*. Elsewhere in the document, other events (such as cancellation, 18.2.1.2 *"When the last destructor function returns, thread shall be terminated."*) are said to cause the thread to terminate as well.

There is a possible interpretation that solves this apparent conflict, but this interpretation depends on the precise meaning of the term *thread termination* and is very subtle. The interpretation is that the text in 16.1.2.2 refers only to the case when *main()* returns normally, while the text in 16.1.4.2 discusses the case when the main thread terminates other than by returning from *main()* or calling *exit()*. That is, the value zero is returned if the main thread terminates by calling *pthread_exit()*, cancellation, or by some other means for which the effect is not to immediately terminate the whole process.

16.15 Values of Thread-Ids and Thread Objects

Class: [IMP]

Reference: 16.1.6

Issue:

Can two distinct values of *pthread_t* designate the same thread object? If so, does *pthread_equal()* return a non-zero (i.e. true) value when two such values are compared?

Discussion:

Assuming that *pthread_equal()* indeed returns non-zero in the above case, the first question is largely academic and should not affect portable applications.

16.16 Erroneous Use of *pthread_equal()*

Class: [ERR], [UND]

Reference: 16.1.6.3 [LINES: 458 (D8)]

Issue:

The effect of calling *pthread_equal()* with invalid values for *t1* or *t2*.

16.17 Erroneous Use of *pthread_once()*

Class: [ERR], [UND]

Reference: 16.1.7.2 [LINES: 492-493 (D8)]

Issue:

What is the behavior of *pthread_once()* if *once_control* was not properly initialized (by PTHREAD_ONCE_INIT), or if it has an automatic storage duration?

Chapter 17: Thread-Specific Data

This section describes portability issues related to Section 17 of P1003.4a.

17.1 Repeatedly Destroying Thread-Specific Data

Class: [MAY]

Reference: 17.1.1.2 [LINES: 733-740 (D9)]

Reference: 17.1.2.2 [LINES: 193-195 (D8)]

Issue:

Whether or not PTHREAD_DESTRUCTOR_ITERATIONS is honored. (i.e. how many times will the destructor functions be called?)

Discussion:

This seems like a minor portability issue, that in practice should not affect real applications. It is only useful when allocating more thread-specific data from within a destructor of such data, a dubious practice easily avoided or protected against. At the extreme, it may lead to loss of memory, in a very hard-to-detect sequence.

17.2 Using Deleted Keys

Class: [ERR], [UND]

Reference: 17.1.2.2 [LINES: 189-191 (D8)]

Reference: 17.1.3.2 [LINES: 241-242 (D8)]

Issue:

The effect of using invalid or deleted keys.

Discussion:

This is a user error situation. In order to prevent this from occurring, a portable application must provide a synchronization layer around the code that deletes keys. In fact, the situation is even more complex: Unless the logic of the application ensures that there is a natural place to delete keys [1], the entire use of thread-specific data has to be encapsulated in a layer that duplicates all the access functions. This is needed to

[1] such as at the end of the program, but then, what is the point in deleting the keys in the first place?

ensure that no threads still use a key after it has been deleted (e.g. some mechanism of reference-count will be needed).

A special case exists when the invalid or deleted key is used for the *pthread_key_delete()* call itself. Based on the discussion above, it seems that some protection mechanism will be needed. If one indeed assumes the existence of such a mechanism, then there is no need for more than one thread to delete keys, and thus, this special case is also a user error situation.

17.3 Thread Specific Keys Manipulation Functions as Macros

Class: [MAY]

Reference: 17.1.2.2 [LINES: 196 (D8)]

Issue:

Are *pthread_setspecific()* and *pthread_getspecific()* implemented as macros?

Discussion:

To a C-language programmer, such an implementation dependency is rarely noticeable. The only difference is the inability to treat the functions as independently-callable entities. A language binding will not be able to call these functions directly if they may be implemented as macros.

Chapter 18: Thread Cancellation

This section describes portability issues related to Section 18 of P1003.4a.

18.1 Transfer of Control Out of Cancellation Scopes

Class: [ERR], [UND]

Reference: 18.1 [LINES: 107-109 (D8)]

Reference: 18.1.3 [LINES: 304 (D8)]

Reference: 18.2.3.2 [LINES: 508-513 (D8)]

Issue:

The effect of using longjmps, goto's, etc. to leave user-defined cancellation scopes.

Discussion:

Officially, such behavior is not supported by P1003.4a. If one accepts this limitation, then this becomes a simple error situation.

However, this limitation is considered a major handicap of this facility. For example, for thread cancellation to be useful for the implementation of the Ada Asynchronous Transfer of Control, it would require the ability to jump out from the middle of cancellation handlers back into the task code, thereby modifying the target task flow of control.

18.2 Asynchronous Cancellation and Cleanup Handler Management

Class: [ERR], [IMP]

Reference: 18.1 [LINES: 111-113 (D8)]

Issue:

The effect of using *pthread_cleanup_push()* or *pthread_cleanup_pop()* when the cancelability type in effect is PTHREAD_CANCEL_ASYNCHRONOUS.

Discussion:

This is a simple error situation. It seems logical to establish the cleanup handlers only when the code is protected from cancellation.

18.3 Implementation-Defined Cancellation Points

Class: [MAY]

Reference: 18.1.2 [LINES: 175-228 (D8)]

Issue:

P1003.4a lists here the functions that *"may"* be implemented as cancellation points. The question is whether they are really implemented as such.

Discussion:

This is a major portability issue. If these functions are indeed implemented as cancellation points, applications must ensure that they are protected from cancellation during the call. On the other hand, when the length of non-cancelable regions is important, portable applications cannot rely on these functions to be cancellation points. Note that the standard does not require implementations to document their choice. In fact, whether or not any of these functions is a cancellation point can change dynamically over time.

18.4 Races Between Events and Cancellation Requests

Class: [UNS]

Reference: 18.1.2 [LINES: 242-245 (D8)]

Issue:

The sequence of events when the cancel request and the waited-upon event come at the same time.

Discussion:

We assume that the following text discusses only the case of PTHREAD_-CANCEL_ASYNCHRONOUS.

> *"However, if the thread is suspended at a cancellation point and the event that it is waiting for occurs before the cancellation request is acted upon, it is unspecified whether the cancellation request is acted upon or whether the request remains pending and the thread resumes normal execution."*

If this is indeed so, then there is no problem since this situation is an unavoidable race condition against which the application has to protect.

18.5 Implementation-Defined Async-Cancel-Safe Functions

Class: [MAY]

Reference: 18.1.5 [LINES: 337-338 (D8)]

Issue:

This paragraph lists the functions that are required to behave as async-cancel-safe. The question is are there any other (implementation-defined) async-cancel safe functions?

Discussion:

Portable applications should not depend on other functions being async-cancel safe. However, since P1003.4a defines a very small number of such functions, and since such a property (async-cancel safety) is very useful, it is likely that implementations will provide more of these, and applications are likely to unwittingly take advantage of this, leading to a lack of portability. Note that implementations are not required to document their choice here.

18.6 Push/Pop as Macros

Class: [MAY]

Reference: 18.2.3.2 [LINES: 503-507 (D8)]

Issue:

Whether *pthread_cleanup_pop()* and *pthread_cleanup_push()* are implemented as macros?

Discussion:

To a C-language programmer, such an implementation dependency is rarely noticeable. The only difference is the inability to treat the function as an independently-callable entity. A language binding will not be able to use these functions anyhow since they are tied to a specific C-language scope.

18.7 Error Conditions for *pthread_cleanup_push()*

Class: [IMP]

Reference: 18.2.3.4 [LINES: 523-524 (D8)]

Issue:

No error conditions are defined for *pthread_cleanup_pop()* and *pthread_cleanup_push()*.

Discussion:

What happens when errors do occur? (For example, when there is not enough space on the stack for more handlers to be pushed.) This is simply a hole in the definition. For applications that demand high reliability, this omission might prove to be a problem and a source of non-portability.

Chapter 19: Summary and Conclusions

This report has attempted to enumerate all the explicit and implicit implementation dependences that currently exist in P1003.4a. Draft 9 of this proposed standard was examined since it was the most recent one to be published at the time of completing this report.

As has been clearly demonstrated, many such dependences exist. Some of these appear to be intentional. Others seem to be due to unintended imprecision in the wording of the document. Even though in most cases, the intent of the text can be surmised, this is inappropriate for a widely-used standard.

When a standard is introduced in a new area such as this one, it is not always clear how it will be used and how it will evolve. Thus, while many of the issues itemized in this report are considered minor at present and we expect they will have little or no relevance to a real application, they may prove to be important later on for one group of users or another. Furthermore, the sheer number of such issues identified (more than a hundred) clearly adds to the magnitude of the problem.

We believe that when the scope of the implementation dependence is local (such as setting an invalid value in an output parameter when a POSIX function fails), its effect is minor since a local solution usually suffices. However, we have found several issues that affect the overall design of the application, and hence limit the design choices available to portable applications.

As can be expected, the major areas of concern are process scheduling and signal handling. For a real-time application, the importance of clear and portable semantics in these areas cannot be over emphasized.

We hope that the findings of this report will draw attention to the problems that remain in P1003.4a, and will help in improving it before the standard is approved. For the issues that remain unresolved, the list assembled here can be used as a check-list for users and vendors. As a result, better vendor-documentation will be available and convergence of implementations (in these areas) will be encouraged.

Annex A
(informative)
Bibliography

{1} *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language].*, IEEE (1990).

{2} *Threads Extension for Portable Operating Systems, Draft 9*, IEEE (May, 1994).

{3} *Realtime Extension for Portable Operating Systems, Draft 14*, IEEE (March, 1993).

{4} *Standardized Application Environment Profile— POSIX Realtime Application Support (AEP), Draft 6.0*, IEEE (June, 1994).

{5} *The Gnu NYU Ada Translator*, available by anonymous `ftp`, from New York University.

{6} E.W. Giering and T.P. Baker. POSIX/Ada real-time bindings: Description of work in progress. In *Proceedings of the Ninth Annual Washington Ada Symposium*. ACM, July 1992.

{7} E.W. Giering and T.P. Baker. Using POSIX threads to implement Ada tasking: Description of work in progress. In *TRI-Ada '92 Proceedings*, pages 518–529. ACM, November 1992.

{8} E.W. Giering, Frank Mueller, and T.P. Baker. Implementing Ada 9x features using POSIX threads: Design issues. In *TRI-Ada '93 Proceedings*, pages 214–228. ACM, September 1993.

{9} E.W. Giering III and T.P. Baker. Ada 9X asynchronous transfer of control: Applications and implementation. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

Alphabetic Topical Index

active priority... 4
alarm... 20
allocation domains... 7
asctime_r... 10
Asynchronous IO... 23
async-signal safety... 15
base priority... 4
Bibliography... 61
C Language-Specific... 24
cancellation points... 7
Categories... 3
Clocks and Timers... 45
Contact Information... 2
contention scopes... 7
Critical Portability Issues... 5
ctime_r... 10, 25
Definitions... 12
detachstate... 14, 51
EDEADLK... 44
EINTR... 16, 19–20, 24, 27
EINVAL... 51
EPERM... 8
error handling... 4–5, 10, 12, 14, 16, 18, 24–27, 29–30, 32, 46, 49, 52, 54–55, 57
ESRCH... 51
Execution Scheduling... 36
exit... 53
fault tolerance... 7, 10, 27, 29–30, 50, 52, 57
Files and Directories... 22
flockfile... 24
funlockfile... 24
implementation defined... 3, 8, 14, 18–19, 26, 28, 40–43, 49
implementation dependence, implicit... 3, 9–10, 15, 17, 20, 24–25, 27, 30–31, 36, 43, 46, 48, 50, 53, 57, 59
integration conflict... 3, 15
integration hole... 4, 9, 23, 35, 39, 45–46
Introduction... 1
  subclause ... 5, 8, 36
localtime_r... 25
longjmp... 25
main... 53
Memory Mapping... 35
memory model... 12
Message Passing... 46
Most Serious Problems... 6
mq_close... 46
msync... 35
Non Section-Specific Issues... 8
option, explicit... 3, 12–13, 21, 41
option, implicit... 3, 10, 12, 25, 28, 33, 49, 55–56, 58–59
options... 6, 13
permission... 8
_POSIX_THREAD_PRIORITY_SCHEDULING... 8
POSIX_THREADS... 40
POSIX_THREADS_PRIORITY_SCHEDULING... 40
_POSIX_THREAD_THREADS_MAX... 52
priority... 4
  asynchronous I/O ... 23
  base ... 4
  execution ... 4
  inherited ... 4
  interactions ... 7
  original ... 4
  process versus thread ... 7
  queuing ... 7
  temporary ... 4
Process Environment... 21
pthread_attr_destroy... 49
PTHREAD_CANCEL_ASYNCHRONOUS... 33, 57–58
PTHREAD_CANCEL_DEFERRED... 33
PTHREAD_CANCEL_ENABLE... 33
pthread_cleanup_pop... 57, 59
pthread_cleanup_push... 57, 59
pthread_condattr_destroy... 28
pthread_cond_destroy... 28
pthread_cond_signal... 6, 31
pthread_cond_timedwait... 32–33
pthread_cond_wait... 32–33
pthread_create... 8, 52
PTHREAD_CREATE_JOINABLE... 14
PTHREAD_DESTRUCTOR_ITERATIONS... 55
pthread_detach... 51
pthread_equal... 53–54
pthread_exit... 48, 52–53
pthread_getspecific... 56
pthread_join... 48, 51
pthread_key_delete... 56
pthread_mutexattr_destroy... 28
pthread_mutex_destroy... 28
pthread_mutex_lock... 30–31
pthread_mutex_setprioceiling... 44
pthread_mutex_trylock... 30
pthread_once... 54
PTHREAD_ONCE_INIT... 54
PTHREAD_PRIO_INHERIT... 43
PTHREAD_PRIO_PROTECT... 43

PTHREAD_SCOPE_PROCESS... 39, 41
PTHREAD_SCOPE_SYSTEM... 39–41
pthread_setschedparam... 4, 37
pthread_setspecific... 56
pthread_sigmask... 17
pthread_t... 48–49, 53
pthread_yield... 40
Purpose... 1
readdir_r... 22
SCHED_FIFO... 32, 42–43
SCHED_OTHER... 43
SCHED_RR... 42–43
sched_setparam... 39
sched_setscheduler... 39
scheduling... 36–38, 44
    mutex effects ... 27, 31
    permission to set parameters ... 8
semantic hole... 3, 14, 18, 24, 33, 36–38, 40, 51, 53
sigaction... 16
SIGALARM... 20
siglongjmp... 25
signal side-effects... 6, 16, 26, 33
signals... 16–20
Signals... 14
sigpending... 17
sigprocmask... 17
sigqueue... 20
sleep... 20
spurious wakeups... 6
strtok_r... 10
Structure... 1
Summary and Conclusions... 60
Synchronization... 26
Terminology for Priorities... 4
Terms and Conventions... 3
Thread Cancellation... 57
Thread Management... 48
thread safety... 12, 24
Thread-Specific Data... 55
undefined behavior... 3, 14, 16, 18, 24–27, 29, 32, 52, 54–55, 57
unspecified... 3, 12, 15–17, 19, 22, 39, 47–49, 52, 58
virtual processors... 9