IEEE Standards Interpretations for IEEE Std 1003.1c™-1995 IEEE Standard for Information Technology--Portable Operating System Interface (POSIX(R)) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)

These are interpretations of IEEE Std 1003.1c-1995.

Interpretation Request #3
Topic: Miscellaneous **Relevant Clauses:** Many

1) Clause 16.1.1.2, page 141 D10, lines 79-83 A default value for the stackaddr attribute is not specified. The behavior is specified if an application wants to specify their own stack. It does not specify what to do if they want the implementation to create a stack for them. What is the default value of this attribute? NULL to signify the user wants the implementation to allocate stacks?

2) Clause 16.1.1.2, page 141 D10, lines 74-78 A default value of the stacksize attribute is not specified. The behavior is specified if an application wants to specify their own stack size. It does not specify the value of stacksize if the user wants the implementation to use a default stacksize. What is the correct stacksize to return as a default value when the user has not specified a stacksize? Some implementations specify a default value of 0 for a default stack. Other implementations specify a default value of PTHREAD_STACK_MIN. This confusion can lead to source code portablity problems.

3) Clause 13.5.1.2, page 121 D10, lines 316-332 A default value for the inheritsched attribute is not specified. The default behavior needs to be specified for portable applications or it could result in differing behavior across platforms. What is the default value for inheritsched?

4) Clause 13.5.1.2, page 122 D10, lines 333-342 A default value for the schedpolicy attribute is not specified. What is the default value for this attribute?

5) Clause 13.5.1.2, page 122 D10, lines 353-357 A default value for the schedparam attribute is not specified. What is the default value for this attribute? Implementations are already providing different values (drastically) which will effect portability. Some implementations consider the default value to be NULL; others return an actual value for this attribute as the default value. This case is not merely different default values, but actual differences in what this attribute represents. This impacts portability. What is the default value/behavior of this attribute?

6) Clause 17.1.2.2, page 166 D10, lines 192-195 This paragraph states that calling pthread_setspecific() from within a destructor function may result in lost storage or infinite loops. This can be a real problem. POSIX.1c has just stated that it is impossible to use thread-specific data in an application. A portable and correctly behaving application cannot rely on calling pthread_setspecific() from within a destructor function. What makes thread-specific data impossible to use is the fact that in pthread_key_create() it is stated that when a thread terminates, any non-NULL TSD values which have destructors will have the destructor called for them. The problem here is that the destructor function is called in a loop (potentially forever -- so a portable application has to assume forever) until the key value is NULL. However, according to pthread_setspecifc() a portable and correctly behaving application has no way possible to change the key value to a NULL value. Portable application have to rely on "worst-case" guaranteed behavior. According to the definitions of pthread_setspecific() and pthread_key_create() an application cannot reliably make use of thread-specific data with destructor functions or its thread will end up in an infinite loop. Could you please clarify the intended behavior of these functions? Obviously there must be something missing here or this looping behavior for destructor functions would not have been added since an application must assume it results in an infinite loop.

7) Clause 13.3.1.2, page 114 D10, lines 30-36 Clause 13.3.3.2, page 114 D10, lines 42-48 According to the new rules for scheduling, the sched_setparam() and sched_setscheduler() are not useless in the presence of multithreaded applications. The only effect these functions have is on the child process [from fork()] of the target process. An implementation may cause something to happen to the process scope threads, but that's all. This provides a huge hole for system administrators. These functions have a process d parameter. This means they were intended so that a process could control the policy and priority of another process. If that wasn't the case, there wouldn't have been a pid as a parameter. Up until now a system administrator could control a process if it was getting too much or too little time on the system. Runaway processes with high priority could be handled. With the new behavior, there is no way a sysadmin (or any process) can control the behavior of a multithreaded process. If some event happens requiring a change in the policy/priority of the MT process, only the process itself can do it. The worst part is that if one thread in the process goes out of control while a high priority SCHED_FIFO, no one can control it. A sysadmin cannot do anything to lower that thread's priority (thread IDs are not guaranteed to be known outside of the system). Is this the actual intended behavior?????

8) Clause 13.6.1.2, page 128-129 D10, lines 581-587 What is the default value of the protocol attribute?

9) Clause 13.6.1.2, page 128-129 D10, lines 578-580, 595-606 PTHREAD_PRIO_PROTECT This states that a) the priority of the locking thread shall raise its priority to the mutex prioceiling and b) that prioceiling must be a valid priority for SCHED_FIFO. What happens if the locking thread is not SCHED_FIFO? Chances are pretty good that the prioceiling value is not a valid priority for the scheduling policy of the thread. Even if the thread is SCHED_RR the value may not be valid. Do these functions imply that you must also change the scheduling policy of the locking thread to SCHED_FIFO? If so, what happens if the system defines SCHED_FIFO to have lower priorities than say SCHED_RR and the thread is under SCHED_RR?

10) Clause 13.6.1.2, page 128-129 D10, lines 590-594 PTHREAD_PRIO_INHERIT This states that the blocking thread shall raise the priority of the thread owning the mutex to equal that of the blocking thread. Only the priority is being changed here. What happens if the threads are in different scheduling policies? The new priority may not be valid in that scheduling policy. Is it assumed that these functions also change the policy of the thread if they are different?

11) Clause 3.1.3.2, page 27 D10, lines 84-94 This function allows an application to essentially install cancellation type handlers to guarantee that the proper state is maintained in the child process after a fork. What is supposed to happen with allocated thread-specific data in the child process? These functions are, in effect, executed by the thread calling fork. If the other threads have allocated a lot of thread-specific data, there is no way for the process to release that memory. The child has immediately inherited a memory leak when using TSD. Is this intended?

12) Clause 3.3.6.2, page 39 D10, lines 492-494 This section states that sigpending() returns the signals pending on “either” the process or the thread. The way stated, an implementation is allowed to return process pending signals or thread pending signals. An application cannot portably use and rely on what this function does because of “either”. What the actual intent to say something more like “returns the union of the signals pending on the process and the calling thread”?

13) Clause --None in P1003.1c D10, corresponding section in IEEE Std 1003.1b-1995 is 14.2.2.2 The behavior for timer_create() specifies what happens when a sigevent structure of type SIGEV_NONE or type SIGEV_SIGNAL is passed to the function. POSIX.1c does not specify what the behavior of this function is if the sigevent structure is for SIGEV_THREAD. The most complicated part is when sigevent specifies SIGEV_THREAD but the timer is a reloading timer so that it continually expires. What is supposed to happen with timers and SIGEV_THREAD? A thread is to be created when the timer expires? What happens with reloading timers which continually expire? Does only one thread get created and from then on an overrun count is incremented? Since these threads are de-

tached, you have no way of knowing this thread terminates in order to stop incrementing the count and create a new thread on the next timer expiration.

Interpretation Response

1. Stack address: The standard is clear that the default value of the stackaddr attribute is "unspecified". A conforming system is free to choose any value for this field and a conforming application must correctly handle any value. If the attribute is supported, the application can specify the placement of the stack by using an attribute object for thread creation and the function pthread_attr_setsetstacksize. The interpretations committee believes that this was the intent of the working and balloting groups. Due to the complexities of O/S and compiler interactions, the expectation was that the thread implementation will wish to reserve to itself the ability to determine the placement and size of the stack. The rationale for adding the functions to set the fields was derived from the needs of real-time systems and embedded environments where the management of memory is often handled by the application writer. An application writer wishing to force a particular stack address will need to be vary cautious since different systems, processors, O/Ss, and compilers will varying in the amount of memory required for the stack of an application. This issue is analogous to the stack creation in the exec functions, where there are no particular requirements on the location or size of the stack in the new process image.

2. Stack size: The standard is clear that the default value of the stacksize attribute is "unspecified". A conforming system is free to choose any value for this field and a conforming application must correctly handle any value. If the attribute is supported, the application can specify the value of the stack size by using an attribute object for thread creation and the function pthread_attr_setsetstacksize. The interpretations committee believes that this was the intent of the working and balloting groups. Due to the complexities of O/S and compiler interactions, the expectation was that the thread implementation will wish to reserve to itself the ability to determine the placement and size of the stack. The rationale for adding the functions to set the fields was derived from the needs of real-time systems and embedded environments where the management of memory is often handled by the application writer. An application writer wishing to force a particular stack size will need to be vary cautious since different systems, processors, O/Ss, and compilers will varying in the amount of memory required for the stack of an application. This issue is analogous to the stack creation in the exec functions, where there are no particular requirements on the location or size of the stack in the new process image.

3. Inheritsched The standard is clear that the default value of the inheritsched attribute is "unspecified". A conforming system is free to choose any value for this field and a conforming application must correctly handle any value. The application can specify the value of inheritsched by setting it in the attribute object using the function pthread_attr_setinheritsched. The interpretations committee believes that this was the intent of the working and balloting groups. Due to the wide variety and uses of O/Ss implementing the standard, the expectation is that the implementation will have a default that makes

sense in its context.

4. schedpolicy The standard is clear that the default value of the schedpolicy attribute is "unspecified". A conforming system is free to choose any value for this field and a conforming application must correctly handle any value. The application can specify the value of the schedpolicy by setting it in the attribute object using the function pthread_attr_setschedpolicy. The interpretations committee believes that this was the intent of the working and balloting groups. Due to the wide variety and uses of O/Ss implementing the standard, the expectation is that the implementation will have a default that makes sense in its context.

5. schedparams The standard is clear that the default value of the schedparams attribute is unspecified. A conforming system is free to choose any value for this field and a conforming application must correctly handle any value. The application can specify the value of schedparams by setting it in the attribute object using the function tthread_attr_setschedparam. The interpretations committee believes that this was the intent of the working and balloting groups. Due to the wide variety and uses of O/Ss implementing the standard, the expectation is that the implementation will have a default that makes sense in its context.

6. pthread_setspecific The standard is clear in defining the circumstances under which lost storage of infinite loops may occur as a result of using thread specific data. The statement in 17.1.2.2, lines 72-74 that "calling pthread_setspecific() from a destructor may result in lost storage or infinite loops" is clarified by the description of destructor behavior in 17.1.1.2, lines 26-33. In particular, the circumstances under which an infinite loop or lost storage might occur are described, those being when {PTHREAD_DESTRUCTOR_ITERATIONS} iterations of destructor calls have been made and non-NULL key values remain. As such, calling pthread_setspecific() specifying a NULL value is always safe (even in destructors) since it will not result in such storage loss or infinite loops. Interpretation request number 8 raised a related issue -- asking under what circumstances a thread-specific data value is set to NULL during destructor calls. The standard currently only describes one way -- an explicit call to pthread_setspecific(key, NULL). The interpretations committee believes that it was the intent of the working and balloting groups that the thread-specific data value associated with a key should automatically set to NULL before the destructor is called in order to prevent infinite loops. Otherwise each destructor function would have to somehow determine the key for which it was invoked and do a pthread_setspecific(key, NULL) in order to prevent infinite loops.

7. Can't set thread scheduling parameters from another process The standard is clear that it does not define any interfaces to provide this function. The interpretations committee believes that this was the intent of the working and balloting groups in order to allow the widest possible types of implementations and, specifically, library level implementations would not be able to expose such a system call to another process.

8. protocol The standard is clear that the default value of the protocol attribute is "un-

specified". A conforming system is free to choose any value for this field and a conforming application must correctly handle any value. The application can specify the value of the protocol by setting it in the attribute object using the function pthread_mutexattr_setprotocol and then using the attribute object for the creation of a mutex. The interpretations committee believes that this was the intent of the working and balloting groups.

9. PTHREAD_PRIO_PROTECT The standard is clear on pages 305-6 lines 701-705 that the process executes at the higher of its priority or that of prioceiling. On pages 287-8, the model of scheduling is clear that, conceptually, there is an ordering for all possible priority values, independent of scheduling policy. In the case that the effective priority range of Sched-RR is higher than that of Sched-FIFO, then the mechanisms provided to guard against priority inversion would likely be unsuccessful, should any Sched_RR threads use the mutex. The interpretations committee believes that this was the intent of the working and balloting groups. Note also that the standard does not require that the priority_ceiling value be a valid priority level for the class of a locking thread. A thread's priority is not affected by its inheritance, only its execution. See also Interpretation #10.

10. PTHREAD_PRIO_INHERIT The standard is clear on pages 287-8 that the specification of a scheduling policy is in the context of a conceptual model that requires specifying the relationship between the different polices. The system is to execute the thread with the highest effective priority independent of scheduling policy. To achieve this, it may be necessary for a thread's effective priority to be set to a value outside its policy's priority range.

11. Fork Handlers The standard is clear that memory allocated as thread-specific data for threads other than the forking thread might be leaked in the child process if not freed or otherwise accounted for at fork time since only a copy of the thread that called fork() will exist in the child process. This is the intended behavior. Three mechanisms that applications can use to eliminate this leakage are described in the rationale in B.3.1.3. (1) Perform an exec() call. (2) Use the pthread_atfork() prepare handler to clean up any state associated with other threads before the fork() is executed. (3) Use the pthread_atfork() child handler to clean up any state associated with other threads after the fork() is executed.

12. Signal value returned by sigpending() The standard is clear that the set of signals returned by sigpending() consists of all of those signals that are blocked from delivery and are pending on either the process or the calling thread. Equivalently, this is the union of the following two sets of signals: 1. The set of signals that are blocked from delivery and pending on the process. 2. The set of signals that are blocked from delivery and pending on the calling thread.

13. SIGEV_THREAD The standard is clear on page 74 lines 696-698 that each execution of the signal handler shall be executed in an environment as if it were the start routine of a new thread. The interpretations committee notes that the language in 14.2.2.2

specifying the notification semantics for timer expiration was changed by IEEE Std 1003.1i-1995 to explicitly refer to 3.3.1.2. It is unspecified as to whether multiple signals would be executed sequentially or in parallel: conforming systems are free to do either and conforming applications shall handle the case of parallel execution. The interpretation committee believes that this was the intent of the working and balloting groups in order to allow a wide range of implementations.

Rationale for Interpretation
None.