N3765: Thread-safe signals handling

Document #:	N3765
Date:	2025-10-20
Project:	Programming Language C
Reply-to:	Niall Douglas
	<s_sourceforge@nedprod.com>

This paper revises [N3540] *Modern signals handling* to incorporate committee feedback. This paper proposes composable signal handling which is thread-safe and optionally thread-local for the standard C library. Its reference implementation library can be found at `https://github.com/ned14/wg14_signals`.

The reference implementation library is designed to be easily mergeable into any standard C library as well as used standalone. It requires a C11 compiler to compile, but only a C89 compiler to use. It works on POSIX and Microsoft Windows on at least the x64 and AArch64 architectures (and likely many more). It works well in the Fil-C guaranteed memory safe C compiler.

Performance is reasonable: `thrd_signal_invoke()` costs 16 to 20 nanoseconds; threadsafe global signal handlers cost 8 to 20 nanoseconds depending on platform and architecture.

Changes since N3540:

- `struct thrd_raised_signal_info`'s `raw_info` and `raw_context` are no longer `void *`, and instead the new typedefs `thrd_raised_signal_info_siginfo_t` and `thrd_raised_signal_info_context_t` alias the platform-specific types (e.g. `struct __siginfo` or `struct ucontext_t`).
- `thrd_signal_func_t` and `thrd_signal_recover_t` no longer are pointer types.
- `modern_*()` functions renamed to `threadsafe_*()`.
- All the Returns sections now use full sentences.
- Use 'implementation-defined' instead of 'implementation defined'.
- All use of 'will' replaced.
- All use of 'must' replaced.
- All use of 'should' replaced.
- Type `sigset_t` is now required to be complete and assignable.
- The `sig*` originally had failure-success wording to match those of POSIX, which is to say vague and imprecise. I have tightened those up to match major POSIX implementations, but be aware it could theoretically break some POSIX implementation somewhere, particularly around use of signal numbers `>= NSIG` (note that `NSIG` is intentionally not defined by POSIX, see https://www.austingroupbugs.net/view.php?id=741).
- Added definition of *async-signal-safe*, and applied that to functions where it made sense.
- Replaced doxygen marked up comments with normal comments. Mention of system specific types moved into 'e.g.' clauses so they become examples of what might be there, rather than any inference of a requirement.
- I tried my best to improve the descriptions of `threadsafe_signals_install` and `threadsafe_signals_uninstall` by moving everything into a single header paragraph under signal handling.

One request was that the term 'thread-safe' ought to be explicitly defined in normative wording. The C++ standard text does not define it – it assumes it does not need defining, which seems fair to me as it is as well understood as 'loop' is nowadays. The C++ standard uses 'thread-safe' in two forms:

1. Unqualified, in which case it means 'thread safe with respect to everything else'.
2. Qualified e.g. 'This function is thread-safe with respect to `get_tzdb_list().front()` and `get_tzdb_list().erase_after()`.' which means concurrent use of only those functions are thread-safe, and concurrent use of any function outside that subset is not.

I have examined all uses of 'thread-safe' and qualified any I thought necessary.

Contents

1 Overview

This paper takes a conservative approach to modernising signal handling, and fixes these specific problems only:

1. `signal()` (and indeed POSIX `sigaction()`) are not suited for programs containing multiple threads, as installing and removing signal handlers is not threadsafe.

2. `signal()` (and indeed POSIX `sigaction()`) are not suited for programs containing shared libraries, as dynamically loading and unloading a shared library cannot install and remove signal handlers easily i.e. they don't compose well in modern software architecture.

3. There is no ability to install thread local signal handlers so a signal caused by one thread can be handled locally, and it not affect other threads of execution.

4. Finally, `_Thread_local` may, or may not, be async signal handler safe depending on platform. As the previous functionality needs to store thread-local state which is safe to use from within a signal handler, we might as well expose that as a public library API for other code to use.

There is a quarter century of existing practice being standardised here:

1. Microsoft Windows Structured Exception Handling implements exactly what this pure library extension to the C standard library would now implement portably. On Microsoft Windows, the reference library is a thin wrap of Microsoft Windows facilities and does little other work.

2. IBM z/OS appears to also implement exactly what this pure library extension to the C standard library would now implement portably. On z/OS, the reference library should be a thin wrap of z/OS APIs and would do little other work[1].

On POSIX, it exclusively uses `sigaction()` and does not require any new functionality in the C standard library. The reference implementation internally uses `setjmp()` and `longjmp()` on POSIX to implement signal recovery, but this paper does not mandate that and an implementation can use any mechanism it likes internally.

It is expected that after this paper is merged, I will propose a second paper *Even more modernised signal handling* to WG14 which takes a less conservative approach than this paper does. It will enable some signal handlers to execute arbitrary code not unencumbered by async signal handling restrictions. This is possible because every major platform provides proprietary APIs to provide signal handling by a background kernel thread, so this is another area ripe for standardisation seeing as this facility is ubiquitous across the major platforms. This is why some of the functions in this proposal take a `version` argument. That proposal will overlay this proposal, and can be considered entirely an extension to this proposal which may be separately accepted or rejected by WG14.

Finally, there is a very great deal of existing code out there using existing signals, and the API has been complicated by needing to be as good a citizen as possible to existing code. We allow users to partially opt-in, partially opt-out, and entirely opt-out. We also allow existing signal handling

---

[1]My thanks to Rajan for illuminating the z/OS documentation to me. I unfortunately have no access to such a system to port this library to z/OS, however from reading its documentation, such a port should be trivial.

code to call into this modern signals handling, so people can still have their code be the first layer of signal handling, and then vector into this code after.

It is proposed that for one major C standard that these facilities shall be opt-in with `signal()` marked deprecated. The following C standard these facilities would become explicit opt-out.

2   Notes about the proposed wording

- The current C standard references POSIX 1993. There is an argument that it ought to be updated to at least the latest ISO POSIX release, which is POSIX 2008. Or, considering that POSIXs after 2008 are published under the IEEE, POSIX 2024.

  However I think such a change should be its own proposal paper. I am unaware of anything in this specific paper which requires anything newer than in POSIX 1993. For example, all the POSIX `sig*()` functions being standardised into C were in POSIX 1993.

- Can we fix the weird async signal safety of `atomic_is_lock_free()` but ONLY if its argument is an atomic? Can't we make it guaranteed always async signal safe?

  Also, why isn't that function guaranteed thread safe? If runtime code has to be executed to determine if the passed type is lock free, that code would not be useful if it were itself racy.

3   Proposed wording

This is against `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3485.pdf`. Red strike through is removal, underlined green is addition.

7.14.1 General

The header <`signal.h`> declares ~~a type and two functions and defines several macros~~ types, functions and macros, for handling various *signals* (conditions that may be reported during program execution).

The ~~type defined is~~ types defined are

`typedef thrd_raised_signal_error_code_t`

which is an implementation-defined native error code type.

`union thrd_raised_signal_info_value`

which is the union of `intptr_t int_value` and `void *ptr_value`.

> [*Note:* This differs from POSIX `union sigval` by making the integer type `intptr_t` instead of `int`. – end note]

`typedef thrd_raised_signal_info_siginfo_t`

which is an implementation-defined signal information type.

`typedef thrd_raised_signal_info_context_t`

which is an implementation-defined context type.

struct `thrd_raised_signal_info`

which shall include at least the following members:

```
1  // The signal raised
2  int signo;
3
4  // The system specific error code for this signal, e.g. the 'si_errno'
5  // code (POSIX) or 'NTSTATUS' code (Windows)
6  thrd_raised_signal_error_code_t error_code;
7
8  // Memory location which caused fault, if appropriate
9  void *addr;
10
11 // A user-defined value
12 union thrd_raised_signal_info_value value;
13
14 // The system specific information e.g. 'siginfo_t *' (POSIX)
15 // or 'PEXCEPTION_RECORD' (Windows). Can be null if the
16 // value was not supplied, or was supplied as null.
17 thrd_raised_signal_info_siginfo_t *raw_info;
18
19 // The system specific context e.g. 'ucontext_t' (POSIX) or
20 // 'PCONTEXT' (Windows). Can be null if the
21 // value was not supplied, or was supplied as null.
22 thrd_raised_signal_info_context_t *raw_context;
```

typedef union `thrd_raised_signal_info_value` (`thrd_signal_func_t`)
(union `thrd_raised_signal_info_value`)

which is the type of the function invocable by `thrd_signal_invoke`().

enum `thrd_signal_decision_t`

which shall include at least the following members:

```
1  // We have decided to do nothing
2  thrd_signal_decision_next_decider
3
4  // We have fixed the cause of the signal, please resume execution
5  thrd_signal_decision_resume_execution
6
7  // Thread local signal deciders only: reset the stack and local
8  // state to entry to 'thrd_signal_invoke()', and call the recovery
9  // function.
10 thrd_signal_decision_invoke_recovery
```

typedef enum `thrd_signal_decision_t` (`thrd_signal_decide_t`)(struct `thrd_raised_signal_info`

which is the type of the function invocable by `thrd_signal_invoke`() and globally installed signal deciders to decide how to handle a raised exception.

`sig_atomic_t`

which is the (possibly volatile-qualified) integer type of an object that can be accessed as an atomic entity, even in the presence of asynchronous interrupts.

`sigset_t`

which is an implementation-defined complete and assignable type able to represent a set of signals on this platform.

The macros defined are

`SIG_DFL`

`SIG_ERR`

`SIG_IGN`

which expand to constant expressions with distinct values that have type compatible with the second argument to, and the return value of, the `signal` function, and whose values compare unequal to the address of any declarable function; and the following,.

The following which expand to positive integer constant expressions with type int and distinct values that are the signal numbers, each corresponding to the specified condition:

`SIGABRT` abnormal termination, such as is initiated by the abort function, which is of *synchronous signal* category

`SIGFPE` an erroneous arithmetic operation, such as zero divide or an operation resulting in overflow, which is of *synchronous signal* category

`SIGILL` detection of an invalid function image, such as an invalid instruction, which is of *synchronous signal* category

`SIGINT` receipt of an interactive attention signal, which is of *asynchronous nondebug signal* category

`SIGSEGV` an invalid access to storage, which is of *synchronous signal* category

`SIGTERM` a termination request sent to the program, which is of *asynchronous nondebug signal* category

Add new paragraph after 7.14.1.4:

Only the functions in the standard library listed below are required to be *async-signal-safe*; all other functions can be *async-signal-unsafe*. When a signal interrupts an *async-signal-unsafe* function and the signal-catching function calls an *async-signal-unsafe* function, the behavior is undefined. The following functions are required to be *async-signal-safe*:

- the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free,

- the `atomic_is_lock_free` function with any atomic argument,

- the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the signal function results in a `SIG_ERR` return, the object designated by `errno` has an indeterminate representation, or

- any function within this standard explicitly described as *async-signal-safe*.

Add new paragraph after the paragraph above:

The `signal` function globally installs a single signal handler for the program. It is not a thread safe operation, nor does it compose well with library code as the single existing handler always gets overwritten, and handler invocation cannot be chained.

The `threadsafe_signals_install` function enables an alternative signal handling mechanism which may (depending on implementation) install a signal handler for the program which implements thread-safe signal handling. If you choose to use this alternative signal handling mechanism, then the following sequence occurs on signal raise:

1. If there are thread locally installed signal deciders with a signal set matching the signal number, invoke each of those signal deciders on the calling thread in order of most recently installed for that thread. Each decider function is called with a pointer to a populated `struct thrd_raised_signal_info`, with its `value` set to the `value` as was specified when that decider was installed. If any decider function returns:

   - `thrd_signal_decision_resume_execution`: execution is resumed.

   - `thrd_signal_decision_invoke_recovery`: the stack and local state are restored to what they were when that thread local decider was installed, and the recovery function as specified at that time will be invoked to implement recovery from the signal raise for that thread.

   - `thrd_signal_decision_next_decider`: the next decider in the sequence is invoked.

2. If there are no thread locally installed signal deciders, or every thread locally installed signal decider returned `thrd_signal_decision_next_decider`, the globally installed signal deciders with a signal set matching the signal number are invoked in order of most recently installed with `callfirst = true`, and then in order of most recently installed with `callfirst = false`. Each decider function is called with a pointer to a populated `struct thrd_raised_signal_info`, with its `value` set to the `value` as was specified when that decider was installed. If any decider function returns:

   - `thrd_signal_decision_resume_execution`: execution is resumed.

   - `thrd_signal_decision_next_decider`: the next decider in the sequence is invoked.

   - `thrd_signal_decision_invoke_recovery`: it is unspecified what occurs.

It is permitted for a signal decider to never return. Signal deciders shall meet the requirements of a well defined signal handler as defined in the `signal` function.

It is implementation-defined what happens if every signal decider returns `thrd_signal_decision_next_decider`: a default handling action would be expected. For example, on POSIX systems which permit the retrieval of the signal handler installed before the `threadsafe_signals_install` function's signal handler was installed, a reasonable default action might be to invoke that previous signal handler.

[*Note:* It should be noted that on some implementations, the thread-safe alternative signal handling mechanism is the native implementation and the `signal` function is emulated using that native system. On such implementations, `signal` based code will always be second tier to the thread-safe alternative signal handling mechanism, and they will define what happens if all handlers installed choose to not handle a signal raise. – end note]

7.14.1.1 The `sigemptyset` function

Synopsis

```
1  #include <signal.h>
2  int sigemptyset(sigset_t *set);
```

Description

Calling this function is thread-safe apart from other operations concurrently acting on *set*, and is *async-signal-safe*.

The set of signals pointed to by *set* shall be set to the empty set as defined by the implementation.

Returns

This function always returns zero for compatibility with POSIX.

7.14.1.2 The `sigfillset` function

Synopsis

```
1  #include <signal.h>
2  int sigfillset(sigset_t *set);
```

Description

Calling this function is thread-safe apart from other operations concurrently acting on *set*, and is *async-signal-safe*.

The set of signals pointed to by *set* shall be set to the full set as defined by the implementation.

Returns

This function always returns zero for compatibility with POSIX.

7.14.1.3 The `sigaddset` function

Synopsis

```
1  #include <signal.h>
2  int sigaddset(sigset_t *set, int signo);
```

Description

Calling this function is thread-safe apart from other operations concurrently acting on *set*, and is *async-signal-safe*.

Signal number *signo* shall be added to the set of signals pointed to by *set*, if it is not already set in which case nothing is done.

If *set* was not originally initialised using either `sigemptyset` or `sigfillset`, behaviour is undefined.

Returns

This function always returns zero for compatibility with POSIX.

7.14.1.4 The `sigdelset` function

Synopsis

```
1  #include <signal.h>
2  int sigdelset(sigset_t *set, int signo);
```

Description

Calling this function is thread-safe apart from other operations concurrently acting on *set*, and is *async-signal-safe*.

Signal number *signo* shall be removed from the set of signals pointed to by *set*, if it is not already unset in which case nothing is done.

If *set* was not originally initialised using either `sigemptyset` or `sigfillset`, behaviour is undefined.

Returns

This function always returns zero for compatibility with POSIX.

7.14.1.5 The `sigismember` function

Synopsis

```
1  #include <signal.h>
2  int sigismember(const sigset_t *set, int signo);
```

Description

Calling this function is thread-safe apart from other operations concurrently acting on *set*, and *async-signal-safe*.

If *set* was not originally initialised using either `sigemptyset` or `sigfillset`, behaviour is undefined.

Returns

If signal number *signo* is set within the set of signals pointed to by *set*, positive one shall be returned. If it is not present, zero shall be returned.

7.14.2.1 The `signal` function

In 7.14.2.1.5:

If the signal occurs other than as the result of calling the `abort` or `raise` function, the behavior is undefined if the signal handler refers to any object with static or thread storage duration that is not a lock-free atomic object and that is not declared with the `constexpr` storage-class specifier other than by assigning a value to an object declared as `volatile sig_atomic_t`, or the signal handler calls any function in the standard library ~~other than~~ not *async-signal-safe.*

— ~~the `abort` function,~~

— ~~the `_Exit` function,~~

— ~~the `quick_exit` function,~~

— ~~the functions in `<stdatomic.h>` (except where explicitly stated otherwise) when the atomic arguments are lock-free,~~

— ~~the `atomic_is_lock_free` function with any atomic argument, or~~

— ~~the `signal` function with the first argument equal to the signal number corresponding to the signal that caused the invocation of the handler. Furthermore, if such a call to the signal function results in a `SIG_ERR` return, the object designated by `errno` has an indeterminate representation.~~

In 7.14.2.1.7:

Use of this function in a multi-threaded program results in undefined behavior. ~~The implementation shall behave as if no library function calls the signal function.~~

7.14.2.2 The `synchronous_sigset` function

Synopsis

```
#include <signal.h>
const sigset_t *synchronous_sigset(void);
```

Description

Calling this function is thread-safe and *async-signal-safe.*

Synchronous signals are those which can be raised by a thread in the course of its execution. This set can include platform-specific additional signals, however at least these standard signals are within this set: `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGSEGV`.

Returns

This function returns a pointer to a set of signals as described above.

7.14.2.3 The `asynchronous_nondebug_sigset` function

Synopsis

```
#include <signal.h>
const sigset_t *asynchronous_nondebug_sigset(void);
```

Description

Calling this function is thread-safe and *async-signal-safe*.

Non-debug asynchronous signals are those which are delivered by the system to notify the process about some event which does not default to resulting in a core dump. This set can include platform-specific additional signals, however at least these standard signals are within this set: `SIGINT`, `SIGTERM`.

Returns

This function returns a pointer to a set of signals as described above.

7.14.2.4 The `asynchronous_debug_sigset` function

Synopsis

```
1  #include <signal.h>
2  const sigset_t *asynchronous_debug_sigset(void);
```

Description

Calling this function is thread-safe and *async-signal-safe*.

Debug asynchronous signals are those which are delivered by the system to notify the process about some debugging event which defaults to resulting in a core dump.

Returns

This function returns a pointer to a set of signals as described above.

7.14.2.5 The `threadsafe_signals_install` function

Synopsis

```
1  #include <signal.h>
2  void *threadsafe_signals_install(const sigset_t *guarded, int version);
```

Description

Calling this function is thread-safe.

For all signals in the signal set `guarded`, a reference count for each shall be as-if incremented. If the as-if reference count was initially zero for that signal number, a system specific installation action for that signal number is performed.

> [*Note:* One standards conforming system specific installation action could be to invoke internal code shared by the implementation of the `signal` function for that signal number, setting an internal signal handler function which implements thread-safe signal handling. The `signal` function itself would not be sufficient, as it does not support signal handler functions which receive `thrd_raised_signal_info_siginfo_t *` and `thrd_raised_signal_info_context_t *`. – end note]

11

Returns

If `version` is not zero, this functions a null pointer and `errno` shall be set to `EINVAL`.

If the installation was unsuccessful, this function returns a null pointer.

If the installation was successful, this function returns a handle to this installation which can be later passed to `threadsafe_signals_uninstall()`.

7.14.2.6 The `threadsafe_signals_uninstall` function

Synopsis

```
1  #include <signal.h>
2  int threadsafe_signals_uninstall(void *handle);
```

Description

Calling this function is thread-safe.

For all signals in the signal set originally installed by the `threadsafe_signals_install()` which returned `handle`, a reference count for each shall be as-if decremented. If the as-if reference count becomes zero for that signal number, a system specific uninstallation action is performed.

> [*Note:* One standards conforming system specific uninstallation action could be to invoke internal code shared by the implementation of the `signal` function for that signal number, restoring the handler which was set before it was replaced by signal handler function which implements thread-safe signal handling. The `signal` function itself would not be sufficient, as it does not support retrieving the existing signal handler. – end note]

Returns

If successful, this function returns zero. If unsuccessful, this function returns a nonzero value.

7.14.2.7 The `threadsafe_signals_uninstall_system` function

Synopsis

```
1  #include <signal.h>
2  int threadsafe_signals_uninstall_system(int version);
```

Description

Calling this function is thread-safe.

If the implementation as-if performs a `threadsafe_signals_install()` by default on program initialization, calling this function restores signal handling to before that specified by `version`.

> [*Note:* In a future C standard, if modern signal handling is enabled by default, calling this function would return signal handling to as it is now i.e. actively downgrade modern signals handling to preceding C standards. – end note]

Returns

If successful, this function returns zero. If unsuccessful, this function returns a nonzero value.

7.14.2.8 The `signal_decider_create` function

Synopsis

```
1  #include <signal.h>
2  void *signal_decider_create(const sigset_t *guarded, bool callfirst,
3                              thrd_signal_decide_t decider,
4                              union thrd_raised_signal_info_value value);
```

Description

Calling this function is thread-safe.

Installs a global signal continuation decider function, which shall be async signal handler safe. See 7.14.1.6 for how global signal continuation decider functions are called.

If `callfirst` is true, installs the function at the top of the list to be called before any other functions currently in the list, otherwise it is installed at an implementation-defined place in the list.

Returns

If unsuccessful, this function returns a null pointer. If successful, this function returns a non-null pointer which can be later passed to the `signal_decider_destroy()` function.

7.14.2.9 The `signal_decider_destroy` function

Synopsis

```
1  #include <signal.h>
2  int signal_decider_destroy(void *handle);
```

Description

Calling this function is thread-safe.

Uninstalls a previously installed global signal continuation decider function.

Returns

If successful, this function returns zero. If unsuccessful, this function returns a nonzero value.

7.29.5.9 The `thrd_signal_invoke` function

Synopsis

```
1  #include <signal.h>
2  union thrd_raised_signal_info_value
3  thrd_signal_invoke(const sigset_t *signals,
4                     thrd_signal_func_t guarded,
5                     thrd_signal_recover_t recovery,
6                     thrd_signal_decide_t decider,
7                     union thrd_raised_signal_info_value value);
```

Description

Calling this function is thread-safe and *async-signal-safe*. The `decider` function shall be *async-signal-safe*.

Installs a thread local signal continuation decider function, recording the current stack and local state such that it can be restored later if this recovery function is ever invoked. See 7.14.1.6 for how thread local signal continuation decider functions are called.

Returns

If no signal was raised, this function returns the value returned by `guarded`. If a signal was raised and a signal decider initiated recovery, this function returns the value returned by `recovery`.

7.29.5.10 The `thrd_signal_raise` function

Synopsis

```
1  #include <signal.h>
2  bool thrd_signal_raise(int signo,
3                         thrd_raised_signal_info_siginfo_t *raw_info,
4                         thrd_raised_signal_info_context_t *raw_context);
```

Description

Calling this function will be thread-safe and *async-signal-safe*.

As-if invoke `raise`, but with the added information `raw_info` and `raw_context`.

> [*Note:* For your information the reference implementation library does not raise a real signal on POSIX, but simulates raising one instead because there is no other way to pass in a custom `siginfo_t` and `ucontext_t`. If it reaches the end of all lists, it calls the signal handler which was installed before modern signals was installed. On Microsoft Windows, it does actually raise a real Win32 exception as for those you can specify a custom `EXCEPTION_RECORD` and `CONTEXT`. As both thread local and globally installed signal handlers are directly installed with Windows, it will perform its default action when it runs out of handlers. Suggestion to implementers: I think it would be preferable if a real signal was initially raised where possible, then debuggers get notified. You may be able to persuade your standard C library to implement this e.g. on Linux one can use syscall `rt_tgsigqueueinfo`. – end note]

Returns

It is implementation-defined if this function ever returns, but if it does, this function returns true if at least one signal decider installed under this facility was invoked.

7.29.1 Introduction

`tss_async_signal_safe`

which is a complete object type that holds an identifier for an *async-signal-safe* thread-specific storage pointer;

struct `tss_async_signal_safe_attr`

which shall include at least the following members:

```
1  // Create an instance
2  int (*create)(void **dest);
3
4  // Destroy an instance
5  int (*destroy)(void *v);
```

7.29.6.5 The `tss_async_signal_safe_create` function

Synopsis

```
1  #include <threads.h>
2  int tss_async_signal_safe_create(tss_async_signal_safe *val,
3                                   const struct tss_async_signal_safe_att) *attr);
```

Description

Creates an *async-signal-safe* thread-specific storage pointer. A copy of `attr` will be taken, this describes function pointers later called to create and destroy instances of the thread-specific storage. The object pointed to by `val` shall be set to a value that uniquely identifies the newly created instance.

Returns

This function returns `thrd_success` if successful and `thrd_error` if unsuccessful.

7.29.6.6 The `tss_async_signal_safe_destroy` function

Synopsis

```
1  #include <threads.h>
2  int tss_async_signal_safe_destroy(tss_async_signal_safe val);
```

Description

Destroys a previously created *async-signal-safe* thread-specific storage pointer.

All thread-specific storage pointers associated with this instance will be destroyed using the original `attr->destroy()` upon the successful return of this function.

Returns

This function returns `thrd_success` if successful and `thrd_error` if unsuccessful.

7.29.6.7 The `tss_async_signal_safe_thread_init` function

Synopsis

```
1  #include <threads.h>
2  int tss_async_signal_safe_thread_init(tss_async_signal_safe val);
```

Description

Creates the thread-specific storage pointer for the calling thread by invoking the original `attr->create()`.

It is implementation-defined if the thread-specific storage pointer created has the original `attr->destroy()` called upon it on thread exit, if that occurs before the call to `tss_async_signal_safe_destroy()`.

Returns

This function returns `thrd_success` if successful and `thrd_error` if unsuccessful.

7.29.6.8 The `tss_async_signal_safe_thread_get` function

Synopsis

```
1  #include <threads.h>
2  void *tss_async_signal_safe_get(tss_async_signal_safe val);
```

Description

Calling this function will be *async-signal-safe.*

`tss_async_signal_safe_thread_init()` shall have been called on the calling thread beforehand, in which case the thread-specific storage pointer created at that time is returned; otherwise it is undefined what occurs.

Returns

This function returns the thread-specific storage pointer for the calling thread.

7.25.5.1 The `abort` function

The `abort` function causes abnormal program termination to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. An implementation-defined form of the status *unsuccessful termination* is returned to the host environment by means of the function call `raise(SIGABRT)`. Calling this function is *async-signal-safe.*

7.25.5.5 The `_Exit` function

The `_Exit` function causes normal program termination to occur and control to be returned to the host environment. No functions registered by the `atexit` function, the `at_quick_exit` function, or signal handlers registered by the `signal` function are called. The status returned to the host environment is determined in the same way as for the `exit` function. Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined. Calling this function is *async-signal-safe.*

7.25.5.7 The `quick_exit` function

The `quick_exit` function causes normal program termination to occur. No functions registered by the `atexit` function or signal handlers registered by the signal function are called. If a program calls the `quick_exit` function more than once, or calls the `exit` function in addition to the `quick_exit` function, the behavior is undefined. If a signal is raised while the `quick_exit` function is executing, the behavior is undefined. Calling this function is *async-signal-safe.*

4   References

[N2471] Douglas, Niall
  WG14 N2471/WG21 P2069: Stackable, thread local, signal guards
  https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2471.pdf

[N3540] Douglas, Niall
  Modern signals handling
  https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3540.pdf