N3540: Modern signals handling

Document #:	N3540
Date:	2025-04-24
Project:	Programming Language C

Reply-to:	Niall Douglas <s_sourceforge@nedprod.com>

In January 2020 I submitted a paper [N2471] *Stackable, thread local, signal guards* which contained both a C and C++ API implementing composable signal handling which is thread-safe and optionally thread-local. WG21 was not interested in progressing the C++ side of things further, whereas a WG14 straw poll indicated support in principle, so I promised at the time to make a WG14 only edition of the paper.

This took me rather more years than expected, principally because its C-only reference library needed to be written first and there was never enough free time. However, I finally got there and the reference implementation library can be found at `https://github.com/ned14/wg14_signals`.

This 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.

Contents

1 Overview

WG14 discussed N2471 in year 2020, and there was a straw poll in favour of accepting something like N2471 into the IS, which is this paper. 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.

If you examine the reference implementation code, it is fairly trivial. Why I think it is important for WG14 to standardise this is as follows:

1. A single, standard, API is a common lingua franca for all C speaking languages to interact with signal handling, and each other.

2. A single, standard, API lets third party libraries interact with signal handling without causing unpleasant surprise to other code e.g. one bit of code installs a signal handler, another bit of code overwrites the handler.

3. By standardising this API, we can encourage OS kernel maintainers to improve kernel support for thread-aware signal handling.

Soon I will be proposing another paper *Even more modern 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 Proposed wording

This is against https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3467.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]

`struct thrd_raised_signal_info`

which should include at least the following members:

```
int signo;  //!< The signal raised

//! \brief The system specific error code for this signal, the ‘si_errno‘
//! code (POSIX) or ‘NTSTATUS‘ code (Windows)
thrd_raised_signal_error_code_t error_code;

void *addr;  //!< Memory location which caused fault, if appropriate

union thrd_raised_signal_info_value value;  //!< A user-defined value

//! \brief The OS specific ‘siginfo_t *‘ (POSIX) or ‘PEXCEPTION_RECORD‘
//! (Windows)
void *raw_info;

//! \brief The OS specific ‘ucontext_t‘ (POSIX) or ‘PCONTEXT‘ (Windows)
void *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 should include at least the following members:

```
//! \brief We have decided to do nothing
thrd_signal_decision_next_decider

//! \brief We have fixed the cause of the signal, please resume execution
thrd_signal_decision_resume_execution

//! \brief Thread local signal deciders only: reset the stack and local
//! state to entry to ‘thrd_signal_invoke()‘, and call the recovery
//! function.
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 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

7.14.1.1 The `sigemptyset` function

Synopsis

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

Description

Calling this function will be thread safe and async signal handler safe.

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

Returns

Zero if successful, nonzero if unsuccessful.

7.14.1.2 The `sigfillset` function

Synopsis

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

Description

Calling this function will be thread safe and async signal handler safe.

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

Returns

Zero if successful, nonzero if unsuccessful.

7.14.1.3 The `sigaddset` function

Synopsis

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

Description

Calling this function will be thread safe and async signal handler safe.

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

Returns

Zero if successful, nonzero if unsuccessful.

7.14.1.4 The `sigdelset` function

Synopsis

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

Description

Calling this function will be thread safe and async signal handler safe.

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

Returns

Zero if successful, nonzero if unsuccessful.

7.14.1.5 The `sigismember` function

Synopsis

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

Description

Calling this function will be thread safe and async signal handler safe.

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

Returns

A negative number if failure; otherwise the result is success.

7.14.2.1 The `signal` function

- any function within this standard described as *async signal handler safe*,

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 will be thread safe and async signal handler 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

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 will be thread safe and async signal handler 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

A pointer to a set of signals as described above.

7.14.2.4 The `asynchronous_debug_sigset` function

Synopsis

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

Description

Calling this function will be thread safe and async signal handler safe.

Debug asynchronous signals are those which are delivered by the system to notify the process about some event which defaults to resulting in a core dump. This set will be platform-specific signals.

Returns

A pointer to a set of signals as described above.

7.14.2.5 The `modern_signals_install` function

Synopsis

```c
#include <signal.h>
void *modern_signals_install(const sigset_t *guarded, int version);
```

Description

Calling this function will be thread safe.

If appropriate for the platform, for all signals in the signal set `guarded`, a reference count for each will be incremented. If the reference count was initially zero for that signal number, an implementation defined installation action may be performed.

Returns

If `version` is not zero, a null pointer will be returned and `errno` will be set to `EINVAL`.

If the installation was unsuccessful, a null pointer will be returned.

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

7.14.2.6 The `modern_signals_uninstall` function

Synopsis

```c
#include <signal.h>
int modern_signals_uninstall(void *handle);
```

Description

Calling this function will be thread safe.

If appropriate for the platform, for all signals in the signal set originally installed by the `modern_signals_install()` which returned `handle`, a reference count for each will be decremented. If the reference count becomes zero for that signal number, an implementation defined uninstallation action may be performed.

Returns

Zero if successful, nonzero if unsuccessful.

7.14.2.7 The `modern_signals_uninstall_system` function

Synopsis

```c
#include <signal.h>
int modern_signals_uninstall_system(int version);
```

Description

Calling this function will be thread safe.

If the implementation as-if performs a `modern_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

Zero if successful, nonzero if unsuccessful.

7.14.2.8 The `signal_decider_create` function

Synopsis

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

Description

Calling this function will be thread safe.

Installs a global signal continuation decider function, which must be async signal handling safe. If it returns `thrd_signal_decision_resume_execution`, execution is resumed whereas if it returns `thrd_signal_decision_next_decider`, the next global signal continuation decider function in the chain is called.

If true `callfirst` 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

A null pointer if unsuccessful, otherwise 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

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

Description

Calling this function will be thread safe.

Uninstalls a previously installed global signal continuation decider function.

Returns

Zero if successful, nonzero if unsuccessful.

7.29.5.9 The `thrd_signal_invoke` function

Synopsis

```
#include <signal.h>
union thrd_raised_signal_info_value
thrd_signal_invoke(const sigset_t *signals,
                   thrd_signal_func_t guarded,
                   thrd_signal_recover_t recovery,
                   thrd_signal_decide_t decider,
                   union thrd_raised_signal_info_value value);
```

Description

Calling this function will be thread safe and async signal handler safe. The `decider` function must be async signal handler safe.

Record the current stack and local state such that it can be restored later before invoking the recovery function, adding this layer of deciders as the most recent in the stack of deciders for the calling thread. Then invoke `guarded` with `value`. If a signal is raised during the execution of `guarded`, it will be handled as-if `thrd_signal_raise()` for that signal number, signal information and execution context were performed. If no signal is raised during `guarded`, returns the value returned by `guarded`.

Returns

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

7.29.5.10 The `thrd_signal_raise` function

Synopsis

```
#include <signal.h>
bool thrd_signal_raise(int signo, void *raw_info, void *raw_context);
```

Description

Calling this function will be thread safe and async signal handler safe.

If there are thread locally installed signal deciders, call the most recently installed signal decider on the calling thread with a signal set matching `signo` each in turn from the newest to the oldest installed. The decider functions are called with a pointer to a populated `struct thrd_raised_signal_info`, with its `value` set to the `value` as was specified when the decider was installer. If any decider returns `thrd_signal_decision_resume_execution`, this function immediately returns `true`; if it returns `thrd_signal_decision_invoke_recovery`, the stack and local state are restored to what they were when that decider was installed, and the recovery function as specified at the time will be invoked to implement recovery from the signal raise.

If all the thread locally installed signal deciders return `thrd_signal_decision_next_decider`, iterate the list of globally installed signal deciders whose signal set matches `signo`, calling each in turn. If the decider returns `thrd_signal_decision_next_decider`, call the next one in sequence; if it returns

`thrd_signal_decision_resume_execution`, this function immediately returns `true`. It is unspecified what happens if a globally installed signal decider returns `thrd_signal_decision_invoke_recovery`.

It is permitted for a signal decider to never return. Signal deciders must not perform any action which is async signal handler unsafe.

It is implementation defined what happens if every signal decider returns `thrd_signal_decision_next_decider`, some sort of default action would be expected.

> [*Note:* For your information the reference implementation library does not initially 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. – end note]

Returns

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

7.29.1 Introduction

`tss_async_signal_safe`

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

`struct tss_async_signal_safe_attr`

which should include at least the following members:

```
int (*create)(void **dest);  //!< Create an instance

int (*destroy)(void *v);     //!< Destroy an instance
```

7.29.6.5 The `tss_async_signal_safe_create` function

Synopsis

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

Description

Creates an async signal handler 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.

Returns

If it succeeds it sets the object pointed to by `val` to a value that uniquely identifies the newly created instance and returns `thrd_success`; otherwise, `thrd_error` is returned.

7.29.6.6 The `tss_async_signal_safe_destroy` function

Synopsis

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

Description

Destroys a previously created async signal handler 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

`thrd_success` if successful, `thrd_error` if unsuccessful.

7.29.6.7 The `tss_async_signal_safe_thread_init` function

Synopsis

```
#include <threads.h>
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

`thrd_success` if successful, `thrd_error` if unsuccessful.

7.29.6.8 The `tss_async_signal_safe_thread_get` function

Synopsis

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

Description

Calling this function will be async signal handler safe.

`tss_async_signal_safe_thread_init()` must 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

The thread-specific storage pointer for the calling thread.

3 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