P2069R0: Stackable, thread local, signal guards

Document #:	P2069R0
Date:	2020-01-13
Project:	Programming Language C++ SG1 Concurrency study group SG12 Undefined Behaviour study group EWG Incubator study group LEWG Incubator study group WG14-WG21 liaison group

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

A proposal for standard library support for executing a routine, guarding against compiler-unanticipated failure and interruption (i.e. 'signals'), with the possibility of recovering from the failure and continuing execution. This is a high level design, abstracting away for the majority of use cases any need to care about POSIX signal handlers or Win32 structured exception handling.

A reference implementation of the proposed library facility with API documentation can be found at `https://github.com/ned14/quickcpplib/blob/master/include/signal_guard.hpp`. This specific edition has been in production use for over a year at the time of writing (with the previous edition having had many years of use in production), and has proven to be quite popular with some in the C++ ecosystem i.e. it has been lifted and borrowed by quite a few people, because it solves well an ever growing problem (see Introduction). It works well on Android, FreeBSD, MacOS, Linux and Microsoft Windows on ARM, AArch64, x64 and x86.

Contents

1 Introduction

There are three forms of failure handling currently supported in the C++ standard:

1. Anticipated expected failure, which are usually handled with error codes/enums e.g. `std::error_code`. This has the compiler emit code directly into the hot path to handle the failure.
2. Anticipated unexpected failure, which are usually handled with throws of C++ exceptions. This has the compiler emit failure handling into cold path tables, which are traversed by a runtime routine in the assumed unlikely event of a C++ exception being thrown.
3. Unanticipated unexpected failure, where the compiler has quite literally not generated the code to handle such a failure. This presents a unique problem of how to recover from such failure, as the state of parts of the program may be unknowable.

C++ has made much progress on the first two forms of failure handling since its inception in the 1980s. However with respect to the third form, despite multiple unsuccessful attempts by many eminent individuals at big changes, and many successful small improvements tinkering around the edges, in today's C++ standard we are not particularly dissimilar to where we were in the 4th edition of original Unix in 1973 i.e. exactly as we were when C++ was first begun.

It is long overdue that the C and C++ standards modernise their support for handling of the third form of failure of compiler-unanticipated interruption, better known as *signals*. This paper proposes a modernisation which from consultation with the various stakeholders involved, this author believes will satisfy C, C++, POSIX and the other major programming languages.

1.1 Quick summary of the problem

What was sufficient in the 1970s falls down in many ways in the 2020s. To quickly recap the current situation of why we need to modernise this situation for C++ 23:

1. Installing signal handlers on POSIX today is highly prone to surprise because they are global, neither thread aware nor thread safe, and it is impossible for library code to safely modify global signal handlers when it has no idea what other libraries, other kernel threads, or the application, might have done, or currently be doing, to the global signal handlers.

2. Because POSIX is so awful on signals, all other platforms have much saner, proprietary, alternatives, and there is no portable API which works equally everywhere.

3. Even though memory mapped i/o is not currently supported by standard C++, if WG21 chooses one day to adopt `map_handle` from [P1031] *Low level file i/o* (or any other similar proposal), then reading and writing mapped memory may report 'disk full' via raising an unanticipated interruption. We could do with a mechanism for trapping these very recoverable unanticipated failures, and usefully handle such failure.

4. If WG21 chooses to adopt the 'default fail fast' OOM model as proposed by [P0709] *Zero-overhead deterministic exceptions*, this would cause code which works with STL containers configured with the default allocator to become not stack unwindable when OOM by the container's default allocator occurs. Some mechanism of recovering from non-stack-unwindable OOM would therefore be very useful.

5. Testing whether your code correctly terminates the process via `std::terminate()` or `abort()` under various conditions in unit test code is currently a lot of hassle, which is why most codebases don't bother doing this sort of testing. Making this sort of testing convenient would be helpful.

6. The subset of C++ valid to call after a `longjmp()` called by a signal handler is currently very limited: only reads of writes to variables of type `volatile std::sig_atomic_t` written before the signal occurred is well defined C++. This paper seeks to substantially expand that subset of well defined C++ after a signal is handled.

1.2 Proposal specifics

This paper proposes a standard library function for calling a guarded routine in which unanticipated interruption may occur. It is a function named `signal_guard()`, and it works on by setting up guards for the signal mask supplied for the current kernel thread, storing a resumption point using `setjmp()`, executing the guarded code section, and if that experiences an unanticipated failure a `longjmp()` is performed from the signal handler to the resumption point, where an optional cleanup routine may be invoked, followed by exiting the `signal_guard()` function normally. One thus guards the guarded routine from unanticipated interruption, allowing one to recover and carry on efficiently, and without conflicting with other kernel threads, or third party library code.

This design assumes that calling `longjmp()` from within a signal handler is legal. This is required to be the case on POSIX, but may not be the case on other platforms. See later in this paper for a discussion.

One may specify which unanticipated interruptions ought to be guarded for the guarded routine:

- Process abort.
- Undefined memory access.
- Illegal instruction.
- Process interruption.

- Broken pipe.
- Segmentation fault.
- Floating point error.
- C++ out of memory (instead of throwing `std::bad_alloc`).
- C++ termination (somebody has called `std::terminate()`).

This is a subset of what could be available, and WG21 may wish to standardise all of what POSIX provides (see later). However, the semantics of the less common options vary somewhat more in non-POSIX implementations. The list above was chosen precisely because of the common semantics between the major hosted implementations.

One can configure a callable to be called at the exact moment when the interruption occurs, *in-situ*. This callable may be able to recover the problem, and resume execution from the point of interruption by returning `true`. Alternatively, by returning `false`, it will cause execution to `longjmp()` to just before the guarded routine was entered, and to call the previously described cleanup handler.

It is possible to nest guarded sections within other guarded sections for the current kernel thread arbitrarily, and without restriction of depth.

It is possible to thread safely install global handlers with well defined interactions with thread locally installed handlers, and which are fine with dynamic libraries being loaded and unloaded during which they install and uninstall library-specific handlers.

Finally, all this comes with both a C and C++ API, because POSIX and WG14 are very interested in standardising these facilities for all C and POSIX, as everybody recognises that the current situation is non ideal.

In case the above feature set looks familiar to Microsoft Windows programmers, this proposal is actually standardising a subset of Win32 structured exception handling. Indeed, on Microsoft Windows, the reference implementation is trivial, because Windows already implements almost everything for you. I have already run this proposed design past the relevant folk within Microsoft's Visual C++ and kernel teams, and apart from concern about calling `longjmp()` from within a Win32 exception handler (see later), they have no objection to this proposal in principle for implementation on Microsoft Windows.

2 Example of use

The following is taken from the [P1031] *Low level file i/o* reference implementation of proposed `map_handle::write()`, which is working code in production use right now. I have decluttered and reformatted it a little, and added explanatory comments, otherwise it is identical.

```
1 /* This function implements synchronous gather write for map_handle,
2 which is an i/o handle working upon memory mapped storage.
3 Implementation is easy, simply memcpy() each buffer in the gather
4 buffer list into the mapped memory. However, if the disk runs out
```

```
of free space, a SIGBUS or equivalent shall be raised. We want to
trap that, and return it as an errc::no_space_on_device instead.
*/
map_handle::io_result<map_handle::const_buffers_type>
map_handle::write(io_request<const_buffers_type> reqs, deadline /*d*/) noexcept
{
  // const_buffers_type is a span<const_buffer_type>
  // const_buffer_type is a span<const byte>
  // io_request<T> supplies a const_buffers_type list of buffers to
  // gather write, and an offset within the file at which to write them

  // Where in memory we shall be writing to (addr is base of the map)
  byte *addr = _addr + reqs.offset;

  // Clamp the gather write to the end of the map (length is length of the map)
  size_type togo = reqs.offset < _length ? static_cast<size_type>(_length - reqs.offset) : 0;

  /* This signal_guard() function overload takes a bitfield of what
  to guard against, a callable to be guarded, and a callable to be
  called if the guarded callable is aborted. It returns whatever
  the guarded callable, or the cleanup callable, returns, which in
  this case is false for success, and true for failure.
  */
  if(signal_guard(signalc_set::undefined_memory_access,
                  [&]  // The guarded section of code
                  {
                    for(size_t i = 0; i < reqs.buffers.size(); i++)
                    {
                      const_buffer_type &req = reqs.buffers[i];

                      // If this gather buffer's size exceeds that of
                      // the bytes before end of map, truncate the
                      // buffers returned to those actually written.
                      if(req.size() > togo)
                      {
                        memcpy(addr, req.data(), togo);
                        // We wrote togo bytes, not req.size() bytes
                        req = {addr, togo};
                        // Truncate gather list to buffers written
                        reqs.buffers = {reqs.buffers.data(), i + 1};
                        // Return success
                        return false;
                      }
                      memcpy(addr, req.data(), req.size());
                      // Return where the buffer was written to
                      req = {addr, req.size()};
                      // Ensure changes to the updated buffer is visible after signal
                      mem_flush_stores(&req, sizeof(req));
                      addr += req.size();
                      togo -= req.size();
                    }
                    // Return success
                    return false;
                  },
                  [&](const raised_signal_info *info)  // the cleanup handler
                  {
```

```
                    // Retrieve the memory location associated with the failure
                    auto *causingaddr = (byte *) info->addr;

                    // This could be a undefined memory access not involving
                    // this map at all, if so, re-raise it.
                    if(causingaddr < _addr || causingaddr >= (_addr + _reservation))
                    {
                      // Not caused by this map, so re-raise it on this thread
                      thrd_raise_signal(info->signo, info->raw_info, info->raw_context);

                      // POSIX permit signal handlers to return, also the
                      // handler may be set to SIG_IGN, so if undefined
                      // memory access was not handled, abort.
                      abort();
                    }

                    // The guarded routine failed due to undefined memory
                    // access, so return true to cause no_space_on_device
                    // to be returned by the write() function.
                    return true;
                }))
  {
    // If true was returned, we failed due to no space on device
    return errc::no_space_on_device;
  }
  // Otherwise return buffers successfully written
  return reqs.buffers;
}
```

3 Impact on the Standard

There are three major areas in which this proposal would impact the C++ standard.

Currently, the standard requires that no code which could execute non-trivial destructors be present in the guarded section of code: `longjmp()` is permitted over automatic duration C++ objects if, and only if [csetjmp.syn]:

> A `setjmp`/`longjmp` call pair has undefined behavior if replacing the `setjmp` and `longjmp` by catch and throw would invoke any non-trivial destructors for any automatic objects.

This restriction would be preserved if this proposal is adopted.

As unanticipated interruptions may occur at any time, one must only call async signal safe POSIX functions within guarded code, if one is on POSIX. POSIX.2017 requires the following functions to be async signal safe[1]:

[1] https://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html

- _Exit()
- _exit()
- abort()
- accept()
- access()
- aio_error()
- aio_return()
- aio_suspend()
- alarm()
- bind()
- cfgetispeed()
- cfgetospeed()
- cfsetispeed()
- cfsetospeed()
- chdir()
- chmod()
- chown()
- clock_gettime()
- close()
- connect()
- creat()
- dup()
- dup2()
- execl()
- execle()
- execv()
- execve()
- faccessat()
- fchdir()
- fchmod()
- fchmodat()
- fchown()
- fchownat()
- fcntl()
- fdatasync()
- fexecve()
- ffs()
- fork()
- fstat()
- fstatat()
- fsync()
- ftruncate()
- futimens()
- getegid()
- geteuid()
- getgid()
- getgroups()
- getpeername()
- getpgrp()
- getpid()
- getppid()
- getsockname()
- getsockopt()
- getuid()
- htonl()
- htons()
- kill()
- link()
- linkat()
- listen()
- longjmp()
- lseek()
- lstat()
- memccpy()
- memchr()
- memcmp()
- memcpy()
- memmove()
- memset()
- mkdir()
- mkdirat()
- mkfifo()
- mkfifoat()
- mknod()
- mknodat()
- ntohl()
- ntohs()
- open()
- openat()
- pause()
- pipe()
- poll()
- posix_trace_event()
- pselect()
- pthread_kill()
- pthread_self()
- pthread_sigmask()
- raise()
- read()
- readlink()
- readlinkat()
- recv()
- recvfrom()
- recvmsg()
- rename()
- renameat()
- rmdir()
- select()
- sem_post()
- send()
- sendmsg()
- sendto()
- setgid()
- setpgid()
- setsid()
- setsockopt()
- setuid()
- shutdown()
- sigaction()
- sigaddset()
- sigdelset()
- sigemptyset()
- sigfillset()
- sigismember()
- siglongjmp()
- signal()
- sigpause()
- sigpending()
- sigprocmask()
- sigqueue()
- sigset()
- sigsuspend()
- sleep()
- sockatmark()
- socket()
- socketpair()
- stat()
- stpcpy()
- stpncpy()
- strcat()
- strchr()
- strcmp()
- strcpy()
- strcspn()
- strlen()
- strncat()
- strncmp()
- strncpy()
- strnlen()
- strpbrk()
- strrchr()
- strspn()
- strstr()
- strtok_r()
- symlink()
- symlinkat()
- tcdrain()
- tcflow()
- tcflush()
- tcgetattr()
- tcgetpgrp()
- tcsendbreak()
- tcsetattr()
- tcsetpgrp()
- time()
- timer_getoverrun()
- timer_gettime()
- timer_settime()
- times()
- umask()
- uname()
- unlink()
- unlinkat()
- utime()
- utimensat()
- utimes()
- wait()
- waitpid()
- wcpcpy()
- wcpncpy()
- wcscat()
- wcschr()
- wcscmp()
- wcscpy()
- wcscspn()
- wcslen()
- wcsncat()
- wcsncmp()
- wcsncpy()
- wcsnlen()

- wcspbrk()
- wcsrchr()
- wcsspn()
- wcsstr()
- wcstok()
- wmemchr()
- wmemcmp()
- wmemcpy()
- wmemmove()
- wmemset()
- write()

If this proposal is adopted, every relevant standard library function would need to specify in its normative wording whether it guarantees async signal safety. This would need to be done carefully, as when one is not on POSIX, then different lists of permitted versus non-permitted system calls exist, depending on the system in question. For example, on Microsoft Windows, many of the Win32 APIs which appear equivalent to POSIX ones in the list above are NOT async signal safe because they call `malloc()` e.g. `CreateFile()`. If the impact of this on the standard is too great to be feasible, an excellent intermediate stage is for every library function in the standard library to indicate in its normative wording if it must not cause dynamic memory allocation (which must include any userspace component of any system library APIs called) – as potential dynamic memory allocation equals lack of async signal safety.

The last major impact on the standard is that we would need to greatly expand on what state can be written by a guarded section, and what kinds of state can be legally read from after a signal has been recovered from. In the current standard, the only legal kinds of state read are writes to variables of type `volatile std::sig_atomic_t`, which is very limiting. You may have noticed the use of a function `mem_flush_stores()` in the guarded section above. This function comes from WG14 N2436 *Memory region stores flush and reloads force*, which proposes these functions:

```
enum memory_flush
{
  memory_flush_none,    //!< No main memory flushing.

  memory_flush_retain,  //!< Flush modified cache line in CPU out to main
                        //!!< memory, but retain as unmodified in cache.

  memory_flush_evict    //!< Flush modified cache line in CPU out to main
                        //!< memory, and evict completely from all caches.
};

/*! \brief Ensures that reload elimination does not happen for a region of
memory, optionally synchronising the region with main memory.

\return The kind of memory flush actually used.
\param data The beginning of the byte array to ensure loads from.
\param bytes The number of bytes to ensure loads from.
\param kind Whether to ensure loads from the region are from main memory.
\param order The atomic reordering constraints to apply to this operation.

\note ‘memory_flush_retain‘ has no effect for reloads from main memory,
it is the same as doing nothing. Only ‘memory_flush_evict‘ evicts all the
cache lines for the region of memory, thus ensuring that subsequent loads
are from main memory. Note that if the cache line being reloaded is modified,
it will be flushed to main memory before being reloaded, thus destroying
any modified data there. You should therefore ensure that concurrent
actors never modify main memory with modified cache lines in your CPU.
*/
memory_flush mem_force_reload_explicit(volatile char *data,
                                       size_t bytes,
```

```
                                        memory_flush kind,
                                        memory_order order);

/*! \brief The same as ‘mem_force_reload_explicit()‘, but with
‘kind‘ set to ‘memory_flush_none‘, and ‘order‘ set to ‘memory_order_acquire‘.
This does not reload loads from main memory, and prevents reads and writes
to this region subsequent to this operation being reordered to before this
operation.
*/
memory_flush mem_force_reload(volatile char *data,
                              size_t bytes);

/*! \brief Ensures that dead store elimination does not happen for a region of
memory, optionally synchronising the region with main memory.

\return The kind of memory flush actually used.
\param data The beginning of the byte array to ensure stores to.
\param bytes The number of bytes to ensure stores to.
\param kind Whether to wait until all stores to the region reach main memory.
\param order The atomic reordering constraints to apply to this operation.

\warning On older Intel CPUs, due to lack of hardware support, we always execute
‘memory_flush_evict‘ even if asked for ‘memory_flush_retain‘. This can produce
some very poor performance. Check the value returned to see what kind of flush
was actually performed.
*/
memory_flush mem_flush_stores_explicit(volatile const char *data,
                                       size_t bytes,
                                       memory_flush kind,
                                       memory_order order);

/*! \brief The same as ‘mem_flush_stores_explicit()‘, but with
‘kind‘ set to ‘memory_flush_none‘, and ‘order‘ set to ‘memory_order_release‘.
This does not flush stores to main memory, and prevents reads and writes to
this region preceding this operation being reordered to after this operation.
*/
memory_flush mem_flush_stores(volatile const char *data,
                              size_t bytes);
```

`mem_flush_stores()` is basically a `fsync()` for your compiler (and optionally for your CPU as well). It tells the compiler to immediately flush any stores to the region of bytes specified, and not reorder the stores to elsewhere, or perform dead store elimination.

This operation is very useful for signal guarded sections, because we can make it well defined in the standard to read from variables whose representation bytes had `mem_flush_stores()` called upon them, if no further writes between the flush operation and when the unanticipated interruption occurred. This makes practical the scope of code which you can write in a guarded signal section, and which is well defined if a signal is raised.

N2436 had a weak straw poll in favour at the Ithaca WG14 meeting. The concern was not flaws in the proposal, but rather than the proposal was too small to be worth the committee time on standardisation. WG14 asked for a proposed normative wording for N2436, which I have yet to draft (I felt this paper was more important to help WG14 understand a major use case for N2436).

4 Proposed Design

4.1 The C API

Note that this is C2x-targeted, so things like native `bool` types, `nullptr` and `static_assert` are now valid C code.

```
#if defined(__cplusplus)
extern "C"
{
#endif

  /*! \union raised_signal_info_value
  \brief User defined value.
  */
  union raised_signal_info_value {
    int int_value;
    void *ptr_value;
#if defined(__cplusplus)
    raised_signal_info_value() = default;
    raised_signal_info_value(int v)
        : int_value(v)
    {
    }
    raised_signal_info_value(void *v)
        : ptr_value(v)
    {
    }
#endif
  };
#if defined(__cplusplus)
  static_assert(std::is_trivial<raised_signal_info_value>::value, "raised_signal_info_value is not
      trivial!");
  static_assert(std::is_trivially_copyable<raised_signal_info_value>::value, "raised_signal_info_value
       is not trivially copyable!");
  static_assert(std::is_standard_layout<raised_signal_info_value>::value, "raised_signal_info_value
      does not have standard layout!");
#endif

 //! Typedef to a system specific error code type
#ifdef _WIN32
  typedef long raised_signal_error_code_t;  // NTSTATUS
#else
  typedef int raised_signal_error_code_t;   // errno
#endif

  /*! \struct raised_signal_info
  \brief A platform independent subset of `siginfo_t`.
  */
  struct raised_signal_info
  {
    jmp_buf buf;  //!< setjmp() buffer written on entry to guarded section
    int signo;    //!< The signal raised

    //! The system specific error code for this signal, the `si_errno` code (POSIX)
```

```c
  //! or 'NTSTATUS' code (Windows)
  raised_signal_error_code_t error_code;
  void *addr;                           //!< Memory location which caused fault, if appropriate
  union raised_signal_info_value value;  //!< A user-defined value

  //! The OS specific 'siginfo_t *' (POSIX) or 'PEXCEPTION_RECORD' (Windows)
  void *raw_info;
  //! The OS specific 'ucontext_t *' (POSIX) or 'PCONTEXT' (Windows)
  void *raw_context;
};

//! \brief The type of the guarded function.
typedef union raised_signal_info_value (*thrd_signal_guard_guarded_t)(union raised_signal_info_value
    );

//! \brief The type of the function called to recover from a signal being raised in a
//! guarded section.
typedef union raised_signal_info_value (*thrd_signal_guard_recover_t)(const struct
    raised_signal_info *);

//! \brief The type of the function called when a signal is raised. Returns true to continue
//! guarded code, false to recover.
typedef bool (*thrd_signal_guard_decide_t)(struct raised_signal_info *);

/*! \brief Installs a thread-local signal guard for the calling thread, and calls the guarded
function 'guarded'.
\return The value returned by 'guarded', or 'recovery'.
\param signals The set of signals to guard against.
\param guarded A function whose execution is to be guarded against signal raises.
\param recovery A function to be called if a signal is raised.
\param decider A function to be called to decide whether to recover from the signal and continue
the execution of the guarded routine, or to abort and call the recovery routine.
\param value A value to supply to the guarded routine.
 */
union raised_signal_info_value thrd_signal_guard_call(const sigset_t *signals,
                                                      thrd_signal_guard_guarded_t guarded,
                                                      thrd_signal_guard_recover_t recovery,
                                                      thrd_signal_guard_decide_t decider,
                                                      union raised_signal_info_value value);

/*! \brief Call the currently installed signal handler for a signal (POSIX), or raise a Win32
structured exception (Windows), returning false if no handler was called due to the currently
installed handler being 'SIG_IGN' (POSIX).

Note that on POSIX, we fetch the currently installed signal handler and try to call it directly.
This allows us to supply custom 'raw_info' and 'raw_context', and we do all the things which the
signal handler flags tell us to do beforehand [1]. If the current handler has been defaulted, we
enable the signal and execute 'pthread_kill(pthread_self(), signo)' in order to invoke the
default handling.

Note that on Windows, 'raw_context' is ignored as there is no way to override the context thrown
with a Win32 structured exception.

[1]: We currently do not implement alternative stack switching. If a handler requests that, we
simply abort the process. Code donations implementing support are welcome.
 */
```

```
  bool thrd_raise_signal(int signo, void *raw_info, void *raw_context);

  /*! \brief On platforms where it is necessary (POSIX), installs, and potentially enables,
  the global signal handlers for the signals specified by `guarded`. Each signal installed
  is threadsafe reference counted, so this is safe to call from multiple threads or instantiate
  multiple times. On platforms with better than POSIX global signal support, this function does
  nothing.

  ## POSIX only
  Any existing global signal handlers are replaced with a filtering signal handler, which
  checks if the current kernel thread has installed a signal guard, and if so executes the
  guard. If no signal guard has been installed for the current kernel thread, global signal
  continuation handlers are executed. If none claims the signal, the previously
  installed signal handler is called.

  After the new signal handlers have been installed, the guarded signals are globally enabled
  for all threads of execution. Be aware that the handlers are installed with `SA_NODEFER`
  to avoid the need to perform an expensive syscall when a signal is handled.
  However this may also produce surprise e.g. infinite loops.

  \warning This class is threadsafe with respect to other concurrent executions of itself,
  but is NOT threadsafe with respect to other code modifying the global signal handlers.
  */
  void *signal_guard_create(const sigset_t *guarded);

  /*! \brief Uninstall a previously installed signal guard.
   */
  bool signal_guard_destroyl(void *i);

  /*! \brief Create a global signal continuation decider. Threadsafe with respect to
  other calls of this function, but not reentrant i.e. modifying the global signal continuation
  decider registry whilst inside a global signal continuation decider is racy. Called after
  all thread local handling is exhausted. Note that what you can safely do in the decider
  function is extremely limited, only async signal safe functions may be called.

  \return An opaque pointer to the registered decider. `NULL` if `malloc` failed.
  \param callfirst True if this decider should be called before any other. Otherwise
  call order is in the order of addition.
  \param decider A decider function, which must return `true` if execution is to resume,
  `false` if the next decider function should be called.
  \param value A user supplied value to set in the `raised_signal_info` passed to the
  decider callback.
  */
  void *signal_guard_decider_create(const sigset_t *guarded,
                                    bool callfirst,
                                    thrd_signal_guard_decide_t decider,
                                    union raised_signal_info_value value);

  /*! \brief Destroy a global signal continuation decider. Threadsafe with
  respect to other calls of this function, but not reentrant i.e. do not call
  whilst inside a global signal continuation decider.
  \return True if recognised and thus removed.
  */
  bool signal_guard_decider_destroy(void *decider);

#if defined(__cplusplus)
```

```
}
#endif
```

4.2 C++ API

The C++ API reuses the structures from the C API, but reimplements the APIs to not dynamically allocate memory, and thus be fully deterministic for thread local signal guards:

```
//! \brief The signals which are supported
enum class signalc
{
  none = 0,

  abort_process = SIGABRT,           //!< The process is aborting (‘SIGABRT‘)
  undefined_memory_access = SIGBUS,  //!< Attempt to access a memory location which can’t exist
                                     //!< (‘SIGBUS‘)
  illegal_instruction = SIGILL,      //!< Execution of illegal instruction (‘SIGILL‘)
  interrupt = SIGINT,                //!< The process is interrupted (‘SIGINT‘)
  broken_pipe = SIGPIPE,             //!< Reader on a pipe vanished (‘SIGPIPE‘)
  segmentation_fault = SIGSEGV,      //!< Attempt to access a memory page whose permissions disallow
                                     //!< (‘SIGSEGV‘)
  floating_point_error = SIGFPE,     //!< Floating point error (‘SIGFPE‘)

  /* C++ handlers
  On all the systems I examined, all signal numbers are <= 30 in order to fit inside a sigset_t.
  */
  out_of_memory = 32,  //!< A call to operator new failed, and a throw is about to occur
  termination = 33,    //!< A call to std::terminate() was made

  _max_value
};

//! \brief Bitfield for the signals which are supported
BITFIELD_BEGIN_T(signalc_set, uint64_t){
    none = 0,

    //! The process is aborting (‘SIGABRT‘)
    abort_process = (1ULL << static_cast<int>(signalc::abort_process)),
    //! Attempt to access a memory location which can’t exist (‘SIGBUS‘)
    undefined_memory_access = (1ULL << static_cast<int>(signalc::undefined_memory_access)),
    //! Execution of illegal instruction (‘SIGILL‘)
    illegal_instruction = (1ULL << static_cast<int>(signalc::illegal_instruction)),
    //! The process is interrupted (‘SIGINT‘)
    interrupt = (1ULL << static_cast<int>(signalc::interrupt)),
    //! Reader on a pipe vanished (‘SIGPIPE‘)
    broken_pipe = (1ULL << static_cast<int>(signalc::broken_pipe)),
    //! Attempt to access a memory page whose permissions disallow (‘SIGSEGV‘)
    segmentation_fault = (1ULL << static_cast<int>(signalc::segmentation_fault)),
    //! Floating point error (‘SIGFPE‘)
    floating_point_error = (1ULL << static_cast<int>(signalc::floating_point_error)),

    // C++ handlers
    //! A call to operator new failed, and a throw is about to occur
    out_of_memory = (1ULL << static_cast<int>(signalc::out_of_memory)),
```

```
    //! A call to std::terminate() was made
    termination = (1ULL << static_cast<int>(signalc::termination))
} BITFIELD_END(signalc_set)

/*! \brief On platforms where it is necessary (POSIX), installs, and potentially enables,
the global signal handlers for the signals specified by 'guarded'. Each signal installed
is threadsafe reference counted, so this is safe to call from multiple threads or instantiate
multiple times. It is also guaranteed safe to call from within static data init or deinit,
so a very common use case is simply to place an instance into global static data. This
ensures that dynamically loaded and unloaded shared objects compose signal guards appropriately.
On platforms with better than POSIX global signal support, this class does nothing.

## POSIX only
Any existing global signal handlers are replaced with a filtering signal handler, which
checks if the current kernel thread has installed a signal guard, and if so executes the
guard. If no signal guard has been installed for the current kernel thread, global signal
continuation handlers are executed. If none claims the signal, the previously
installed signal handler is called.

After the new signal handlers have been installed, the guarded signals are globally enabled
for all threads of execution. Be aware that the handlers are installed with 'SA_NODEFER'
to avoid the need to perform an expensive syscall when a signal is handled.
However this may also produce surprise e.g. infinite loops.

\warning This class is threadsafe with respect to other concurrent executions of itself,
but is NOT threadsafe with respect to other code modifying the global signal handlers.
*/
class signal_guard_install
{
public:
  explicit signal_guard_install(signalc_set guarded);

  ~signal_guard_install();

  signal_guard_install(const signal_guard_install &) = delete;

  signal_guard_install(signal_guard_install &&o) noexcept;

  signal_guard_install &operator=(const signal_guard_install &) = delete;

  signal_guard_install &operator=(signal_guard_install &&o) noexcept;
};

/*! \brief Install a global signal continuation decider.

This is threadsafe with respect to concurrent instantiations of this type, but not reentrant
i.e. modifying the global signal continuation decider registry whilst inside a global signal
continuation decider is racy. Callable is called after
all thread local handling is exhausted. Note that what you can safely do in the decider
callable is extremely limited, only async signal safe functions may be called.

A 'signal_guard_install' is always instanced for every global decider.
*/
template <class T>
class signal_guard_global_decider
{
```

```
103   public:
104     /*! \brief Constructs an instance.
105
106     \param guarded The signal set for which this decider ought to be called.
107     \param f A callable with prototype `bool(raised_signal_info *)`, which must return
108     `true` if execution is to resume, `false` if the next decider function should be called.
109     \param callfirst True if this decider should be called before any other. Otherwise
110     call order is in the order of addition.
111     */
112     template<class U>
113     requires(std::is_constructible<T, U>::value
114       && requires { std::declval<U>()((raised_signal_info *) 0)); })
115     signal_guard_global_decider(signalc_set guarded, U &&f, bool callfirst);
116
117     ~signal_guard_global_decider() = default;
118
119     signal_guard_global_decider(const signal_guard_global_decider &) = delete;
120
121     signal_guard_global_decider(signal_guard_global_decider &&o) noexcept = default;
122
123     signal_guard_global_decider &operator=(const signal_guard_global_decider &) = delete;
124
125     signal_guard_global_decider &operator=(signal_guard_global_decider &&o) noexcept;
126   };
127
128
129   /*! \brief Call the currently installed signal handler for a signal (POSIX), or raise a Win32
130   structured exception (Windows), returning false if no handler was called due to the currently
131   installed handler being `SIG_IGN` (POSIX).
132
133   Note that on POSIX, we fetch the currently installed signal handler and try to call it directly.
134   This allows us to supply custom `raw_info` and `raw_context`, and we do all the things which the
135   signal handler flags tell us to do beforehand [1]. If the current handler has been defaulted, we
136   enable the signal and execute `pthread_kill(pthread_self(), signo)` in order to invoke the
137   default handling.
138
139   Note that on Windows, `raw_context` is ignored as there is no way to override the context thrown
140   with a Win32 structured exception.
141
142   [1]: We currently do not implement alternative stack switching. If a handler requests that, we
143   simply abort the process. Code donations implementing support are welcome.
144   */
145   bool thrd_raise_signal(signalc signo, void *raw_info = nullptr, void *raw_context = nullptr);
146
147
148   //! \brief Thrown by the default signal handler to abort the current operation
149   class signal_raised : public std::exception
150   {
151   public:
152     //! Constructor
153     signal_raised(signalc code);
154
155     virtual const char *what() const noexcept override;
156   };
157
158   /*! Call a callable `f` with signals `guarded` protected for this thread only, returning whatever
```

```
‘f‘ or ‘h‘ returns.

Firstly, how to restore execution to this context is saved, and ‘f(Args...)‘ is executed, returning
whatever ‘f(Args...)‘ returns if ‘f‘ completes execution successfully. This is usually inlined code
so it will be quite fast. No memory allocation is performed if a ‘signal_guard_install‘ for the
guarded signal set is already instanced. Approximate best case overhead:

- Linux: 28 CPU cycles (Intel CPU), 53 CPU cycles (AMD CPU)
- Windows: 36 CPU cycles (Intel CPU), 68 CPU cycles (AMD CPU)

If during the execution of ‘f‘, any one of the signals ‘guarded‘ is raised:

1. ‘c‘, which must have the prototype ‘bool(raised_signal_info *)‘, is called with the signal which
was raised. You can fix the cause of the signal and return ‘true‘ to continue execution, or else
return ‘false‘ to halt execution. Note that the variety of code you can call in ‘c‘ is extremely
limited, the same restrictions as for signal handlers apply.

2. If ‘c‘ returned ‘false‘, the execution of ‘f‘ is halted **immediately** without stack unwind, the
thread is returned to the state just before the calling of ‘f‘, and the callable ‘g‘ is called with
the specific signal which occurred. ‘g‘ must have the prototype ‘R(const raised_signal_info *)‘
where ‘R‘ is the return type of ‘f‘. ‘g‘ is called with this signal guard removed, though a signal
guard higher in the call chain may instead be active.

Obviously all state which ‘f‘ may have been in the process of doing will be thrown away, in
particular any stack allocated variables not marked ‘volatile‘ will have unspecified values. You
should therefore make sure that ‘f‘ never causes side effects, including the interruption in the
middle of some operation, which cannot be fixed by the calling of ‘h‘. The default ‘h‘ simply throws
a ‘signal_raised‘ C++ exception.

\note Note that on POSIX, if a ‘signal_guard_install‘ is not already instanced for the guarded set,
one is temporarily installed, which is not quick. You are therefore very strongly recommended, when
on POSIX, to call this function with a ‘signal_guard_install‘ already installed for all the signals
you will ever guard. ‘signal_guard_install‘ is guaranteed to be composable and be safe to use within
static data init, so a common use pattern is simply to place a guard install into your static data
init.
*/
template<class F, class H, class C, class... Args>
requires(requires { std::declval<F>()(std::declval<Args>()...) }
  && std::is_constructible<decltype(std::declval<F>()(std::declval<Args>()...)), decltype(std::
      declval<H>()(std::declval<const raised_signal_info *>()))>::value
  && std::is_constructible<bool, decltype(std::declval<C>()(std::declval<raised_signal_info *>()))
      >::value)
inline decltype(std::declval<F>()(std::declval<Args>()...)) signal_guard(signalc_set guarded,
                                                                         F &&f,
                                                                         H &&h,
                                                                         C &&c,
                                                                         Args &&... args);

//! \overload Defaults H to throwing an exception of ‘signal_raised‘
template <class F, class... Args>
requires(requires { std::declval<F>()(std::declval<Args>()...) })
inline decltype(std::declval<F>()(std::declval<Args>()...)) signal_guard(signalc_set guarded,
                                                                         F &&f,
                                                                         Args &&... args);

//! \overload Defaults C to aborting execution of the guarded section, and beginning cleanup
```

```
213   template<class F, class H, class... Args>
214   requires(requires { std::declval<F>()(std::declval<Args>()...) }
215     && std::is_constructible<decltype(std::declval<F>()(std::declval<Args>()...)), decltype(std::
          declval<H>()(std::declval<const raised_signal_info *>()))>::value)
216   inline auto signal_guard(signalc_set guarded, F &&f, H &&h);
```

5 Design decisions, guidelines and rationale

Readers may be surprised to learn that the development of this paper began before all but one of my preceding WG21 papers. It has taken quite a few years to lay the groundwork with the four major stakeholders in signal handling (POSIX, WG14, WG21 and Microsoft) to ensure there would be no immediate vetos. Also, there were multiple rounds of feedback from all four parties regarding design and implementation, which resulted in a large design refactor of the reference implementation. The refactored design then needed at least a year of empirical testing in production before it could be presented here.

5.1 Function taking callable design

It was obvious that attempting to standardise an extension to `try...catch` along the lines of MSVC's `__try` and `__except` extensions implementing thread local signal handling was a non-starter. A lesson was also taken from other attempts in the past to include signals into C++ `try...catch`, which foundered on the severe impact on codegen and optimisation if the compiler must handle unanticipated interruption. It was felt that a better approach would be permitting the compiler to optimise aggressively, and instead allow the programmer to annotate which writes of program state must be well defined to read after signal raise. The compiler can then pessimise only the annotated writes to that state, and nothing else.

The idea of a conventional function taking a guarded callable seemed reasonable, and it had the advantage of working well with Windows' structured exception handling. Making most of it inline-defined in a header file reduced the runtime overhead down to dozens of CPU cycles, which then made it sufficiently low overhead that it could reasonably guard a single pointer dereference, which was a key objective of this proposed design.

5.2 Separate handler install step

For those who have ever had the misfortune of working with them from library code, installing POSIX signals have many problems:

1. Their handlers are installed globally for a process, which creates problems for third party library code.
2. There is only the 'current' signal handler for a signal, which means that 'filtering' signal handlers need to check whether the signal's cause applies to the specific cases they were installed for, and then call the previously installed signal handler.

3. If you install a handler, and then some other code then installs another handler, there is no way to remove your handler because it is now managed by whomever replaced your handler. This makes infeasible installing and removing POSIX signal handlers in dynamically loaded and unloaded shared libraries.
4. Installation and removal of signal handlers is not thread safe.
5. Each thread has a signal mask, which determines which signals can be delivered to it. This means that some signals get delivered to any random thread for which its bit is enabled in that thread's signal mask, which is unhelpful.

For those who have ever used structured exception handling on Microsoft Windows, you will instantly agree that their stackable per-thread approach is the correct way to implement signals. Not what POSIX does.

Implicit in the design presented above for standardisation is effectively stackable per-thread signal handling i.e. what Microsoft Windows does, and indeed on Microsoft Windows, one implements this facility using a trivially simple structured exception handling implementation, as the system already implements everything for us.

On generic POSIX, however – and for the `std::set_terminate()` and `std::set_new_handler()` support on all platforms – one must emulate stackable per-thread signal handling using the global handlers. On generic POSIX, without using platform-specific extensions, this can be done by replacing the global signal handlers with ones which:

1. Check if a `signal_guard` instance for the specific unanticipated failure is present for the calling thread.
2. If so, invokes the guard.
3. If not, calls the previously installed global handler.

This implies that a thread local stack is kept of currently applicable `signal_guard` instances on POSIX, and for the terminate and new global handlers on Windows as well.

Because of this non-trivial setup overhead on POSIX, and the problem of race conditions if you modify the signal handlers outside of program bootstrap, we separate out global handler installation into the `signal_guard_install` class. It would be expected that C++ programs would instance that class somewhere in their static init or their `main()`, however third party libraries can also instance that class in their static init[2], as it is the combined set of `signalc_set` from all the `signal_guard_install` class instances which is actually used.

In other words, it is safe in the proposed design to instance as many `signal_guard_install` objects as you want, and to destruct them in any order. **However** be aware that on POSIX the final `signal_guard_install` instance destruction for a given signal must abort the process if third party code has replaced the handler we installed with another one, as it is not possible to safely deinstall our handler.

[2]Conveniently, all the major dynamic shared library implementations take a global mutex during static init, thus ensuring that only one dynamic shared library can be in the process of being loaded, or unloaded, at any one time.

(Aside: One would hope that if this proposal is standardised, POSIX implementations would internally implement a less broken solution to signal handling, and have this C++ support use that internal implementation instead of the POSIX standard semantics)

5.3 Enabling global signal delivery

If the proposed facility is implemented only using existing POSIX facilities, then `signal_guard_install` globally enables the installed signals for all threads in the process.

Enabling signal delivery for all threads means that the global signal handlers are called from all threads. Obviously, our global signal handler implementation passes on the signal if it cannot find a signal guard instance for the calling thread, however because we are installing a global, filtering signal handler which is active for all threads, we must specify `SA_NODEFER` for the global handler i.e. don't disable the signal during signal handling. This is necessary to avoid deadlock, however the corollary is that if the handler itself causes a signal, it'll loop into itself forever, without termination.

Again, if this proposal were standardised, I would like to hope that POSIX implementations would take the opportunity to substantially refactor how signals are implemented by their C runtime support. I would strongly suggest replicating how Windows implements this, where there are both globally installable AND stackable, per-thread, handlers, with the ability to deinstall a globally installed handler without being the last piece of code to install a handler. The POSIX signal API would then be a subset API for the true, internal, implementation. For more information, see https://docs.microsoft.com/en-gb/windows/desktop/Debug/vectored-exception-handling.

5.4 Use of `longjmp()` to recover from signals

Calling `longjmp()` is legal from signal handlers on POSIX, so that is not a concern. All compilers targeting POSIX therefore generate working code.

Whether it is legal to call `longjmp()` from a Win32 structured exception filter routine is however an open question. About a year ago, I asked Billy O'Neal to connect me up with the relevant people at Microsoft to find out an answer. A discussion by email resulted which lasted more than a week, as people thought through all the reasons why it might not be safe.

The conclusion of that discussion is that it is currently believed that the current MSVC compiler is probably fine with calling `longjmp()` from a Win32 structured exception filter invoked at any pointer within the assembler generated, though it should be stressed that no testing has been performed, and that this is an expert opinion without evidence only. Future MSVC compilers may generate code sequences which are not safe to call `longjmp()` from within, especially as they improve its optimiser. I argued at the time that if GCC and clang manage fine with this, then surely so can MSVC? However as not a compiler implementer, I am not well placed to say with confidence. WG21 feedback is welcome.

5.5 Async signal safe functions on Microsoft Windows

If the committee were to adopt this proposal, the POSIX implementations would have quite a bit of work to improve their signal implementations. This is highly worth doing in any case, but the implementation effort for them is obviously non-trivial.

One would have thought that the impact on Microsoft Windows would be minor given that they have already implemented most of it. However, interestingly there is no internal list of async signal safe functions on Microsoft Windows, and no requirements nor guarantees are made regarding signal safety by any of the teams responsible for those APIs. Many of the Win32 functions are NOT async signal safe, as the userspace portion of their implementation does async signal unsafe things, like take locks, or dynamically allocate or free memory.

So whilst Microsoft would have little code to write in order to implement this proposal, they would have a lot of work to build a list of async signal safe functions, publish it as part of their documentation, and then stick to those guarantees in perpetuity. This is non-trivial implementation effort of a different kind, but significant nonetheless.

6 Frequently Asked Questions

6.1 Why add extra signals to those currently standardised in <csignal>?

<csignal> already defines the Process abort, Illegal instruction, Process interruption, Segmentation fault, and Floating point error signals. To those, this proposal adds:

1. Undefined memory access (SIGBUS)

   Why? Segmentation faults occur when a program tries to access memory whose permissions do not permit that access. Bus errors occur when a program tries to access memory in a way which is not possible e.g. reading or writing an aligned object not at its proper alignment, or an address range for which there are literally no lines in the address bus in the hardware. This is subtly different to SIGSEGV, and I think it worth standardising support for it, especially now that 64-bit addressed systems have become so prevalent (and these often only have 48 address lines in hardware).

2. Broken pipe (SIGPIPE)

   Why? Making use of third party library code may require you to enable delivery of SIGPIPE, because said third party library code does not implement support for EPIPE, leaving you with zero alternative but to use SIGPIPE. This has happened to this author enough times in his career that I think it worth adding to the standard, especially given that SIGPIPE has been in POSIX for nearly forever.

   It should be **stressed** that Broken pipe can be permitted to be meaningless on any particular implementation of C++. It is more a case of 'if your platform might send a Broken Pipe unanticipated interruption, you can use this to recover from it'.

This proposal does **not** propose standardising into C++ these signals currently standardised by POSIX.2017:

- `SIGALRM`
- `SIGCHLD`
- `SIGCONT`
- `SIGHUP`
- `SIGKILL`
- `SIGQUIT`
- `SIGSTOP`
- `SIGTSTP`
- `SIGTTIN`
- `SIGTTOU`
- `SIGUSR1`
- `SIGUSR2`
- `SIGPOLL`
- `SIGPROF`
- `SIGSYS`
- `SIGTRAP`
- `SIGURG`
- `SIGVTALRM`
- `SIGXCPU`
- `SIGXFSZ`

As mentioned earlier, I feel that these signals have poor portability across non-POSIX implementations, or are not worth standardising now. They can always be standardised later, if the need arises.

6.2 What is the interaction with the existing library facility <csignal>?

On POSIX only, signal guard *could* be implemented using <csignal>, apart from the additional signals described above, which are implemented by POSIX in any case. It is highly unlikely, however, that anyone would actually do so when POSIX's `sigaction()` is far superior to `signal()`.

On non-POSIX, I would find it extremely unlikely that anybody would use <csignal> to implement this facility as, in this author's experience, <csignal> implementations have a very low quality of implementation on non-POSIX platforms. To my knowledge, every non-toy non-POSIX system has a proprietary mechanism by which the proposed `signal_guard` function could be completely implemented to a high degree of quality.

7 Acknowledgements

I appreciate that adopting this proposal means an awful lot of work for the committee, and the implementers. It may seem to many that the work involved is not worth the gains, and perhaps this is the case.

I would say however that there was surprising warmth to this proposal from all parties I contacted over the past three years. Everybody is in agreement that current POSIX signals sucks very badly indeed. There was a lot of willingness to go the extra mile if it would help fix POSIX signals. Many people gave freely of their time and sending my approaches through their web of connections to help bring this paper to formal consideration by standards bodies, and I cannot thank you all enough.

I must apologise now for not keeping track of all those who have contributed meaningfully to this paper, either directly, or by connecting me with the people responsible in the various orgs for signal handling. All I can say is that it was one of the first WG21 papers I started, and my paper writing process wasn't as mature as it is now. Because to mention the individuals that I do remember by name would do a disservice to those whose names don't come easily to searches of my email, I have

decided to thank no one by name specifically. However I shall thank those of Microsoft, RedHat, IBM, the Austin Working Group, WG14 and WG21 who helped out.

8 References

[P0709] Herb Sutter,
Zero-overhead deterministic exceptions: Throwing values
https://wg21.link/P0709

[P1031] Douglas, Niall
Low level file i/o library
https://wg21.link/P1031