Synchronization at thread and execution termination

Jens Gustedt

ISO/IEC JTC 1/SC 22/WG14 **N 2391**

v 1

June 7, 2019

Synchronization at thread and execution termination

proposal for integration to C2x

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

Whereas its intent is clear, the C standard lacks clarity concerning synchronization guarantees for the interaction of call backs that may be called during thread termination and during the termination of the whole program execution.

—

This is the "library synchronization" part of a split up of N2329, that was perceived as overwhelming by WG14.

1. PROBLEM DESCRIPTION

C17 establishes several call-back mechanisms that are intended as interfaces for cleanup when either a thread or the whole execution ends:

— destructors for **tss_t**, thread specific storage
— **exit** handlers that are instaured with **atexit**
— **quick_exit** handlers that are instaured with **at_quick_exit**

These are not clearly integrated into the synchronization model, because they do not clearly stipulate sequencing of the different invocations of call-backs among each other, nor their synchronization when they occur in different threads of execution.
Additionally, there is the thread join mechanism with **thrd_join** that guarantees synchronization between the termination of a specific thread and the calling thread. But even here, the synchronization is only clearly specified for the joining thread, not for the terminating thread where it remains unclear how the destructor call-backs are sequenced with respect to thread termination.
We think that what should be done is relatively clear and we didn't hear of misinterpretations which properties should be guaranteed, but we think that some clarification is in order.

2. POSSIBLE SOLUTIONS

2.1. Application synchronization

In principle, the unspecified synchronization properties could be left so, and the burden of ensuring synchronization could be placed on the user code. But then, to ensure proper cleanup of resources that need synchronization, user code would need to add synchronization manually. This would require the use of

— **atomic_thread_fence**, which is only available if the implementation also supports atomics, or
— protection of all handlers by a central **mtx_t**, which then would be locked and unlocked at the beginning and end of each of the destructors and handlers.

2.1.1. Application fences.

For the use with **atomic_thread_fence**:

— User code would have to enforce synchronization of the **exit** and **quick_exit** handlers by establishing explicit synchronization. Since it can't know which handlers are established last (and are thus called first) they'd have to add a call

```
    atomic_thread_fence(memory_order_acquire);
```

At the beginning of each such a handler. When supposing that the current text guarantees that the invocation of the handlers is sequenced, they would not have to add synchronization to the end of the handlers.

— Synchronizing the **tss_t** destructors would necessitate similar acquire fences at the beginning of each destructor, but also release fences at the end. Because the order of destructor invocations is not fixed users can't neither know which of the destructors is called first, nor which is called last.

Besides that these strategies build on the presence of **atomic_thread_fence**, they are tedious and error prone. Subtle synchronization errors could render programming of applications with many threads difficult and insecure.

2.1.2. Application mutex.

For the use with **mtx_t** the user would have to guarantee that all thread termination and all execution of handlers is synchronized through the same mutex. This can *e.g* be achieved by first establishing a dummy handler and a dummy destructor:

```
extern mtx_t sync_mtx;           // Global synchronization utility.
extern tss_t sync_tss;           // Dummy key to enforce call to lock function.
extern once_flag sync_once_flag; // To ensure proper intialization

// Internal functions
extern void sync_last(void);
extern void* sync_dtor(void* p);
extern void sync_once(void);

// API
inline void sync_lock(void) { mtx_lock(&sync_mtx); }
inline void sync_unlock(void) { mtx_unlock(&sync_mtx); }
inline void sync_initializer(void)  {
  call_once(&sync_once_flag, sync_once);
  tss_set(sync_tss, malloc(1));
}

// ***************** implementation *******************************
void sync_last(void) {
  sync_lock();
  sync_unlock();
  mtx_destroy(&sync_mtx);
  tss_delete(sync_tss);
}
void* sync_dtor(void* p) {
  sync_lock();
  free(p);
  sync_unlock();
  return 0;
}
void sync_once(void) {
  mtx_init(&sync_mtx, mtx_plain);
  tss_create(&sync_tss, sync_dtor);
  atexit(sync_last);
  at_quick_exit(sync_last);
}
```

and then to call `sync_initializer()` at the start of each user thread function and to protect each user destructor and each user handler by a pair of calls `sync_lock()` and `sync_unlock()`.
This approach is at least as tedious as the approach with fences above. In addition it has the disadvantage of serializing all destructor calls, even when they are issued for concurrent threads.

2.2. Implementation based solutions

On the other hand, requiring synchronization from the implementation is not much of a burden. Since they know when they call the handlers it is easy for them to add one fence or lock-pair before each of the start and after the end of the call-back procedures.
Because we also think that implementations do something along these veins anyhow, we suggest to go for an implementation based solution.

3. SUGGESTED CHANGES

To make sense for these call-back mechanisms as automatic cleanup procedures, it seems clear that we should require that all call-back invocations should synchronize among each other and with thread and execution termination.

— A destructor should synchronize with the termination of the thread function of the thread for which it is called. Since this concerns only one thread, we just have to insist on proper sequencing between the invocations. Currently the text only talks about an unspecified "order" for these destructor invocations. We propose to use the appropriate terminology and to require that they are "indeterminally sequenced". *See 7.26.5.5 p2.*
— Call-back invocations at the end of program execution (**`exit`** or **`quick_exit`**) should synchronize with all threads that have been properly terminated (**`thrd_exit`** or equivalently **`return`**) and should be sequenced with respect to each other. We introduce two new paragraphs for each of the two functions, *7.22.4.4 p4* and *7.22.4.7 p4*.
— The end of the cleanup mechanism for a particular thread should synchronize with the cleanup mechanism for the whole program execution. *See 7.26.5.5 p4 plus footnote.*

4. IMPACT

The proposed changes are such that they should have no immediate impact on user code or change implementations.
In the very unlikely case that an implementation does not guarantee proper synchronization for the call-backs, yet, they would have to add a modest number of fences surrounding their call-back loops.

Appendix: diffmarks for the proposed changes

Following are those pages that contain diffmarks for the proposed changed against C2x. The procedure is not perfect, in particular there may be changes inside code blocks that are not visible.

Synopsis

1
```
#include <stdlib.h>
int at_quick_exit(void (*func)(void));
```

Description

2 The **at_quick_exit** function registers the function pointed to by `func`, to be called without arguments should **quick_exit** be called.[323] It is unspecified whether a call to the **at_quick_exit** function that does not happen before the **quick_exit** function is called will succeed.

Environmental limits

3 The implementation shall support the registration of at least 32 functions.

Returns

4 The **at_quick_exit** function returns zero if the registration succeeds, nonzero if it fails.

Forward references: the **quick_exit** function (7.22.4.7).

7.22.4.4 The exit function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void exit(int status);
```

Description

2 The **exit** function causes normal program termination to occur. No functions registered by the **at_quick_exit** function are called. If a program calls the **exit** function more than once, or calls the **quick_exit** function in addition to the **exit** function, the behavior is undefined.

3 First, all functions registered by the **atexit** function are called, in the reverse order of their registration,[324] except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the **longjmp** function is made that would terminate the call to the registered function, the behavior is undefined.

4 The beginning of that procedure is a sequence point that synchronizes with the termination of all threads as described for **thrd_exit**. Futhermore, there is a sequence point immediately before and immediately after each of the function calls.

5 Next, all open streams with unwritten buffered data are flushed, all open streams are closed, and all files created by the **tmpfile** function are removed.

6 Finally, control is returned to the host environment. If the value of `status` is zero or **EXIT_SUCCESS**, an implementation-defined form of the status *successful termination* is returned. If the value of `status` is **EXIT_FAILURE**, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.

Returns

7 The **exit** function cannot return to its caller.

7.22.4.5 The _Exit function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void _Exit(int status);
```

[323] The **at_quick_exit** function registrations are distinct from the **atexit** registrations, so applications might need to call both registration functions with the same argument.

[324] Each function is called as many times as it was registered, and in the correct order with respect to other registered functions.

Description

2 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 (7.22.4.4). Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed is implementation-defined.

Returns

3 The **_Exit** function cannot return to its caller.

7.22.4.6 The getenv function

Synopsis

1
```
#include <stdlib.h>
char *getenv(const char *name);
```

Description

2 The **getenv** function searches an *environment list*, provided by the host environment, for a string that matches the string pointed to by name. The set of environment names and the method for altering the environment list are implementation-defined. The **getenv** function need not avoid data races with other threads of execution that modify the environment list.[325)]

3 The implementation shall behave as if no library function calls the **getenv** function.

Returns

4 The **getenv** function returns a pointer to a string associated with the matched list member. The string pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the **getenv** function. If the specified name cannot be found, a null pointer is returned.

7.22.4.7 The quick_exit function

Synopsis

1
```
#include <stdlib.h>
_Noreturn void quick_exit(int status);
```

Description

2 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.

3 The **quick_exit** function first calls all functions registered by the **at_quick_exit** function, in the reverse order of their registration,[326)] except that a function is called after any previously registered functions that had already been called at the time it was registered. If, during the call to any such function, a call to the **longjmp** function is made that would terminate the call to the registered function, the behavior is undefined.

4 The beginning of that procedure is a sequence point that synchronizes with the termination of all threads as described for **thrd_exit**. Futhermore, there is a sequence point immediately before and immediately after each of the function calls.

5 Then control is returned to the host environment by means of the function call **_Exit**(status).

[325)] Many implementations provide non-standard functions that modify the environment list.

[326)] Each function is called as many times as it was registered, and in the correct order with respect to other registered functions.

7.26.5.3 The `thrd_detach` function

Synopsis

1
```
#include <threads.h>
int thrd_detach(thrd_t thr);
```

Description

2 The `thrd_detach` function tells the operating system to dispose of any resources allocated to the thread identified by `thr` when that thread terminates. The thread identified by `thr` shall not have been previously detached or joined with another thread.

Returns

3 The `thrd_detach` function returns `thrd_success` on success or `thrd_error` if the request could not be honored.

7.26.5.4 The `thrd_equal` function

Synopsis

1
```
#include <threads.h>
int thrd_equal(thrd_t thr0, thrd_t thr1);
```

Description

2 The `thrd_equal` function will determine whether the thread identified by `thr0` refers to the thread identified by `thr1`.

Returns

3 The `thrd_equal` function returns zero if the thread `thr0` and the thread `thr1` refer to different threads. Otherwise the `thrd_equal` function returns a nonzero value.

7.26.5.5 The `thrd_exit` function

Synopsis

1
```
#include <threads.h>
_Noreturn void thrd_exit(int res);
```

Description

2 For every thread-specific storage key which was created with a non-null destructor and for which the value is non-null, `thrd_exit` sets the value associated with the key to a null pointer value and then ~~invokes~~ calls the destructor with its previous value. ~~The order in which destructors are invoked is unspecified.~~ These destructor calls are indeterminately sequenced.

3 If after this process there remain keys with both non-null destructors and values, the implementation repeats this process up to `TSS_DTOR_ITERATIONS` times.

4 Following this, the `thrd_exit` function terminates execution of the calling thread and sets its result code to `res`. The sequence point at the end of the execution of the `thrd_exit` function synchronizes with the completion of a successful call, if any, of the `thrd_join` function for the calling thread and with the beginning of all calls of `atexit` or `at_quick_exit` handlers at program termination.[341)]

5 The program terminates normally after the last thread has been terminated. The behavior is as if the program called the `exit` function with the status `EXIT_SUCCESS` at thread termination time.

Returns

6 The `thrd_exit` function returns no value.

7.26.5.6 The `thrd_join` function

[341)] This leaves it unspecified if threads that are terminated by other means than `thrd_exit`, for example by an implementation specific mechanism or because they have not been terminated explicitly before program termination, synchronize with `atexit` or `at_quick_exit` handlers.