Make call_once mandatory

Jens Gustedt

toggle toc

show orig

show new

show both

Make `call_once` mandatory

Jens Gustedt, INRIA, France

org:	ISO/IEC JCT1/SC22/WG14	document:	N2840
target:	IS 9899:2023	version:	1
date:	2021-10-12	license:	CC BY

1. Introduction

C offers several possibilities to attach callbacks to termination events (`atexit`, `at_quick_exit`, `tss` destructors) but only one for initialization, `call_once`. This function entered C11 with the *threads* option and is very usefull in that context, for example for the initialization of static objects with `mtx_t` and `cnd_t` types.

Nevertheless, this function is also very useful in other contexts that have nothing to do with threads, namely for any types that need dynamic initialization to take for example some properties of the platform into account.

Therefore we propose to make this function (and the type and macro) accessible even without threads and to make it mandatory.

2. Changes and additions

2.1. Change the beginning of 7.22 (General utilities <stdlib.h>) p3

> *3 The types declared are `size_t` and `wchar_t` (both described in 7.19); once_flag (described in 7.26)*
> *...*

2.2. Change the beginning of 7.22 (General utilities <stdlib.h>) p4

> *4 The macros defined are NULL (described in 7.19); ONCE_FLAG_INIT (described in 7.26) ...*

2.3. Add a new paragraph 7.22 (General utilities <stdlib.h>) p5

> *5 The function*
>
> ```
> #include <stdlib.h>
> void call_once(once_flag *flag, void (*func)(void));
> ```
>
> *is described in 7.26.2*

3. Impact

These changes do not invalidate user code besides that they add the types `once_flag` and the macro `ONCE_FLAG_INIT` to the header <*stdlib.h*>, which previously had only be reserved if the TU included <*threads.h*>. Otherwise it only adds functionality; the indentifier `call_once` had already been reserved as external since C11.

Changes for implementations are minimal. Those that already have the *threads* option and a monolithic C library have just to add the features to the <*stdlib.h*>. Others that have a separate binary for threads, probably have to do some code movement or add some weak symbol to extend the use to programs that don't use threads.

In a context that does not have threads, implementation of a version that is based on a static integer for `once_flag` objects and polling for its value is straight forward.

3.1. Reference implementation for platforms without threads

Note that `call_once`, as most C library functions, is not guaranteed to be reentrant, see 7.1.4 p4. So for systems that do not have the notition of threads, we also don't have to make provisions for signal handlers.

```
typedef bool once_flag;
#define ONCE_FLAG_INIT false
void call_once(once_flag *flag, void (*func)(void)) {
    if (!*flag) {
        func();
        flag = true;
    }
}
```

3.2. Reference implementation for platforms with C17 atomics and proprietary threads

A version that minimally conforms to the synchronization properties of `call_once` could look as follows:

```
typedef _Atomic(unsigned) once_flag;
#define ONCE_FLAG_INIT 0u
void call_once(once_flag *flag, void (*func)(void)) {
  unsigned actual = atomic_load_explicit(flag, memory_order_acquire);
  if (actual < 2u) {
    switch (actual) {
    case 0u:
      // The very first sets this to 1 and then to 2 to indicate that the function has been
run.
      if (atomic_compare_exchange_strong_explicit(flag, &(unsigned){ 0 }, 1u,
memory_order_relaxed, memory_order_relaxed)) {
        func();
        atomic_store_explicit(flag, 2u, memory_order_release);
        return;
      }
      // we lose and fall through
    case 1u:
      while (atomic_load_explicit(flag, memory_order_acquire) < 2u) {
        // active polling or some sleep if supported
      }
    }
  }
}
```

If a platform does not have C17 atomics but provides atomic extensions (for example in form of low-level atomic instructions) they can easily replace the calls appropriately. The main performance bottleneck for this function is on the “fast path”, namely a call where the value had already been set to `2`, one atomic load and a conditional jump.

4. Question for WG14

Shall we integrate the proposed changes and additions into C23?