

N3961: Atomic waits

Document #: N3961
Date: 2026-08-26
Project: Programming Language C
Reply-to: Niall Douglas
<s_sourceforge@nedprod.com>

For at least a decade now, all the major operating systems have offered the ability to efficiently wait on a value at an address in memory to change. This is provided by these facilities on these major operating systems:

- Linux: kernel `futex`.
- Microsoft Windows: keyed events (NT kernel API) or `WaitOnAddress()` (Win32 API).
- Mac OS: `__unlock_*`() kernel API.
- FreeBSD: `__umtx_*`() kernel API.

C++ 20 via WG21 [P0514R4] added these functions which wrap those proprietary kernel APIs:

- `void atomic_wait(const volatile A *object, C old);`
- `void atomic_wait_explicit(const volatile A *object, C old, memory_order order);`
- `void atomic_notify_one(volatile A *object);`
- `void atomic_notify_all(volatile A *object);`

These are 100% C compatible APIs, so this paper proposes that we also add these functions to the C standard library. For those implementations also with a C++ 20 standard library implementation, implementation for the C standard library editions is trivial; for those implementations without kernel API support, a simple atomic compare exchange spin loop inefficiently provides the right semantics. However, we would recommend instead that implementations either use or borrow from the reference implementation for this proposal which can be found at https://github.com/ned14/wg14_atomic_waits. This is a production quality reference implementation designed to be easily mergeable into any standard C library as well as used standalone. 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: where the atomic type cannot be used directly with the platform API and a *proxy atomic* is used instead, added overhead is around fifty nanoseconds.

With the benefit of hindsight, the C++ atomic wait functions were not ideally designed, so we also propose two new functions for C which perform more predictably:

1. `atomic_wait_expected()`
2. `atomic_notify()`

It is important to not needlessly break C++ compatibility, so this paper proposes keeping API compatibility with the C++ functions – however, newly written code ought to use our two new functions where possible as they lead to more predictable behaviours across platforms. You can, of course, intermix the C++ compatible functions and our new functions arbitrarily if you wish, and it would be expected that the C++ inherited functions would be 100% implemented using our two new functions.

1 The two new atomic wait functions

Our two new proposed functions are as follows:

1. `atomic_wait_expected()` waits until a 32-bit-ish atomic changes and takes an optional timeout; it also can return failures if the OS kernel wait API call fails. One can specify what memory synchronisation ordering occurs on failure as well as when successful.
2. `atomic_notify()` compare exchanges a 32-bit-ish atomic and wakes at least a count of waiting threads; it too can return failures if the OS kernel notify API call fails, which it can on some platforms. One can specify what memory synchronisation ordering occurs on success and failure.

It is worth explaining what is deficient with the C++ atomic wait and notify function design:

1. C++ atomic waits and notifies can never fail

On all platforms atomic wait syscalls CAN fail, and the C++ design retries calling the syscall forever if this occurs.

On Linux and Microsoft Windows the atomic notify syscall can never fail, however on the BSDs and Mac OS they can. If an atomic notify fails one can (a) repeat trying to notify, possibly forever i.e. hang or (b) abort the program. Neither option is attractive to hard code in stone.

Some of the failures returned are actually very useful to know about – particularly the errors morally equivalent to ‘out of memory’ and ‘out of resources’ – so both the new C functions propagate syscall failures.

2. C++ atomic waits throw away the value read

A particular pain point caused by the ABA problem is when an atomic wait returns which means that the atomic changed, but a thread elsewhere overwrites the atomic back to its original value so it looks like the atomic wait has spuriously exited. Debugging this is an absolute pain and it can be a great help to know what value the atomic transiently had to cause the wait to exit. Our functions always return the value read which caused the function to return.

3. C++ atomic waits have no bounded wait option

It should never be possible to sleep a thread without optionally providing a maximum bound in time so one can avoid deadlocks – there is a particularly annoying API design in POSIX where you can either sleep forever for a child process to exit, or you have to spin poll it because there is no standard function for bounded waits. The C++ atomic wait API has this exact same deficiency, so we remedy that here – especially as every platforms’ kernel atomic wait API provides an optional timeout parameter.

4. C++ `atomic` notifies don’t state their memory synchronisation

The C++ standard specifies that there is an ordering of observing the atomic which begins a wait, then the atomic is modified and that happens before the call to the atomic notification. None of this specifies if-whether-how memory is synchronised by the atomic notification.

To that end, our proposed C standardisation of the C++ atomic notify functions requires them to have sequential consistency ordering on memory – this is stricter than their C++ editions. Sequential consistency ordering is often overkill, so our `atomic_notify()` functions lets you explicitly specify memory synchronisation ordering for both success and failure paths, letting one write more optimisable code.

5. C++ `atomic` waits for small atomics are inefficient on most platforms

C++ atomic waits are available for all atomic types, however two major platforms Linux futex and Microsoft Windows keyed events require at least a four byte aligned atomic to directly use the kernel syscall API. This means that for smaller atomics, a *proxy atomic* is used, which adds significant inefficiency as usually an additional hash table indexing step becomes necessary.

This proposal takes the somewhat radical step of restricting our two new functions to 32-bit-ish atomics only as typedefed by `atomic_uint_native_wait_notify_t`, which is the smallest integer larger than or equal to 32 bits which has the least implementation overhead for atomic waits and notifies. There is good reason for the minimum size being thirty-two bits on all platforms ...

6. C++ `atomic_[un]signed_lock_free` is problematic

C++ declares a type `atomic_[un]signed_lock_free` which they define as ‘an integral atomic type that is lock-free and for which waiting/notifying is most efficient’. This is problematic for two reasons:

- (a) C is used more frequently than C++ on architectures without lock free atomics. Certainly insisting on lock free atomics being available is a non-starter for C.
- (b) Their type could be sized anywhere between one and eight bytes depending on implementation.

The latter point matters because atomics are generally used where performance is especially important and where the programmer is highly likely to have carefully laid out their data structures in memory for optimum performance. They might, very reasonably, compile their code on a different implementation for the same hardware expecting similar performance, only to find unpleasant surprise because on the new implementation `atomic_signed_lock_free` is `int8_t` whereas on the previous implementation it was `int32_t`. All their carefully laid out data structures now get the wrong overalignment and wrong packing.

We therefore define `atomic_uint_native_wait_notify_t` to always be at least 32 bits so structures don't unexpectedly disalign. We still permit more than 32 bits for those few platforms without a 32 bit integer – it would be expected that most programmers porting code to a 40 bit integer native architecture would expect by definition all memory layouts to get disrupted.

2 Open design questions

1. Atomic waits are usually used with acquire memory ordering, but there are use cases where other memory orderings can be useful. Should we limit memory ordering to acquire-orientated i.e. other memory orderings would be prohibited?
2. Similarly, atomic notifies are usually used with release memory ordering, though acquire-release for compare exchanges is definitely a common use case. Should other memory orderings be prohibited?
3. `atomic_notify` consists of an atomic compare exchange, and only if that succeeds, the kernel syscall to wake waiting threads is issued. If on the one platform where that kernel syscall can fail did so, then no threads might be woken. Repeating this operation alone would not re-notify the threads as the compare exchange has already occurred, which some might feel is a design hazard.

My own personal opinion: we document that this call can fail. Portable code should take 'some action' e.g. reset the atomic to its expected value, before retrying the notify. What that 'some action' ought to be will depend on use case – we cannot know it in advance.

4. My original design instinct for `atomic_wait_expected` was to flip the logic i.e. if the atomic does NOT equal `wanted`, then block.

The reason why is that such a design allows the thundering herd problem to be eliminated: notifies know what the value was and what it was compare exchanged with, therefore only those waiters waiting on a specific value to appear in the atomic could be woken. This would be considerably more efficient than waking all waiters as the C++ API always does.

Platform support for selective wakes is limited to Linux and Mac OS – on other platforms a wake all would still have to occur. And even there, I'm not sure it adds as much value as one might think – by far and away the most valuable selective wake facility isn't that, but is actually a re-queuing facility: if N threads are blocked on atomic A and all woken threads will immediately block on atomic B, then notifying all on atomic A is very inefficient compared to notifying one and moving N - 1 blocked threads from A to B so they wake as each thread

exits B. Only Linux supports this, and orchestrating the relationship between sequences of wait objects is non-trivial, so I have omitted it from this proposal.

I still raise this open design question however, because in my opinion irrespective of what OS kernel APIs don't currently support, the *correct design* is one which could avoid a thundering herd problem. So maybe WG14 might wish to standardise the correct design, even though OS kernel APIs are currently deficient?

Related documents

number	Title	Authors	Remarks
N3886 (https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3886.pdf)	Working Draft		base for diff, 2026-05-24

Proposed Wording

All changes are relative to N3886.

Legend

Deletions in the shown standard text are as shown [here](#), additions, as shown [here](#). These may render differently according to the style in which the document is shown by your browser, but should always be well distinguishable. In the provided style there are two visual distinctions:

- A high contrast color palette (Okabe and Ito (<https://jfly.uni-koeln.de/color/>)) namely using colors black, orange, teal green and light yellow
- normal text, ~~strike through~~, underlining and typewriter font.

Close to each other proposed changes [resemble like this](#).

7.17.1 Introduction, p5

- Amend p5 as follows:

5 The types include `memory_order`, which is an enumerated type whose enumerators identify memory ordering constraints; `atomic_flag`, which is a structure type representing a lock-free, primitive atomic flag; ~~and several atomic analogs~~ `uint native wait notify t`, which is the `uint_leastN t` type with the smallest N, where N is at least 32, for which atomic waiting and notifying operations have the least overhead on this implementation; and several atomic analogs of integer types.

To 7.17.6 Atomic integer types, p1, Table 7.6, append the row

Type name	C type
<code>atomic_uintmax_t</code>	<code>_Atomic uintmax_t</code>
<code>atomic_uint native wait notify t</code>	<code>_Atomic uint native wait notify t</code>

To 7.17.7.1 General, append:

1 This subclause describes several generic functions that operate on all atomic types other than `atomic_flag`. Also, 7.17.7.6 provides such generic functions for some read-modify-write operations on atomic integer types that are defined in addition to the operators defined in 6.5.3.5, 6.5.4.2 and 6.5.17.

2 The `atomic_wait`, `atomic_wait_explicit`, and `atomic_wait_expected` generic functions are waiting operations, and the `atomic_notify_one`, `atomic_notify_all`, and `atomic_notify_generic` functions are notifying operations. The execution of a waiting operation on an atomic object M can block the calling thread, in which case the thread is blocked on M. A thread blocked on M

becomes unblocked when it is woken up. A wakeup can be caused by a notifying operation on *M* and can also occur for other reasons.

The following new subsections are inserted after 7.17.7.6 and before 7.17.8, and are numbered 7.17.7.7 through 7.17.7.11. The last paragraph of 7.17.7.6 is:

Returns

4 Atomically, the value pointed to by `object` immediately before the effects.

7.17.7.7 The `atomic_wait` generic functions

Synopsis

```
#include <stdatomic.h>
void atomic_wait(const volatile A *object, C expected);
void atomic_wait_explicit(const volatile A *object, C expected, memory_order order);
```

Description

1 Atomically checks the value pointed to by *object* as if by `atomic_load(object)` or `atomic_load_explicit(object, order)`, respectively, and determines whether it compares equal to *expected* as if by the `memcmp` function. If it does not, the function returns immediately. Otherwise, the calling thread is blocked on the atomic object pointed to by *object* until it is woken up; if, when woken up, the value still compares equal to *expected*, the thread is blocked again.

2 NOTE 1 A wakeup can be triggered by a notifying operation on *object* or by an atomic store or exchange to *object*.

3 All side effects to the atomic object *M* that happen before a call *N* to a notifying operation on *M* synchronize with the completion of any waiting operation *W* on *M* that is unblocked by *N*.

7.17.7.8 The `atomic_notify_one` generic functions

Synopsis

```
#include <stdatomic.h>
void atomic_notify_one(volatile A *object);
```

Description

1 If prior to the call a thread has been blocked on the atomic object pointed to by *object* in a waiting operation and has not yet been woken up, at least one such thread is woken up, synchronizing as described in 7.17.7.7.

7.17.7.9 The `atomic_notify_all` generic functions

Synopsis

```
#include <stdatomic.h>
void atomic_notify_all(volatile A *object);
```

Description

1 If prior to the call threads have been blocked on the atomic object pointed to by *object* in waiting operations and have not yet been woken up, all such threads are woken up, synchronizing as described in 7.17.7.7.

7.17.7.10 The `atomic_wait_expected` generic functions

Synopsis

```
#include <stdatomic.h>
int atomic_wait_expected(const volatile A *restrict object, C *restrict expected,
                        const struct timespec *restrict duration,
                        memory_order success, memory_order failure);
```

Description

1 The `sizeof(C)` shall equal the `sizeof(uint native_wait_notify_t)`.

2 Atomically checks the value pointed to by *object* and determines whether it compares equal to **expected* as if by the `memcmp` function. If it does not, the function returns immediately. Otherwise, the calling thread is blocked on the atomic object pointed to by *object* until it is woken up; if, when woken up, the value still compares equal to the initial value of **expected*, the thread is blocked again.

3 NOTE 1 A wakeup can be triggered by a notifying operation on *object* or by an atomic store or exchange to *object*.

4 The total accumulated time during which the calling thread is blocked in this function shall be at least the interval pointed to by *duration* if *duration* is not a null pointer; after that time has elapsed, a timed-out result is returned as soon as is convenient. If *duration* is a null pointer, there are no time limits on when this function returns.

5 All side effects to the atomic object *M* that happen before a call *N* to a notifying operation on *M* synchronize with the completion of any waiting operation *W* on *M* that is unblocked by *N*.

Returns

1 If the function returned immediately because the value did not compare equal to **expected*, or if a timed-out result is returned, the function returns zero and memory is affected according to the value of *failure*. If the calling thread was blocked at least once, the function returns a positive number and memory is affected according to the value of *success*. If the request could not be honored, the function returns a negative value.

2 In all cases, on return **expected* contains the value of **object* that caused the wait to exit; if a timed-out result is returned, **expected* is the same as on entry.

7.17.7.11 The `atomic_notify` generic functions

Synopsis

```
#include <stdatomic.h>
int atomic_notify(volatile A *restrict object, C *restrict expected, C desired,
                 unsigned max_threads_to_wake, memory_order success, memory_order
                 failure);
```

Description

1 The sizeof(C) shall equal the sizeof(uint native wait notify_t).

2 Performs as if by atomic_compare_exchange_strong_explicit(object, expected, desired, success, failure). If the result of that comparison is true, unblocks at least *max_threads_to_wake* threads that are blocked on the atomic object pointed to by *object* in waiting operations.

Returns

1 If the exchange was performed, returns a positive number, possibly one plus the number of threads woken, and memory is affected according to the value of *success*. If the result of the comparison was false, returns zero and memory is affected according to the value of *failure*. If the request could not be honored, the function returns a negative value.

3 Acknowledgements

This paper would not have been written nor have the API design it does without substantial help from Jens Gustedt.

4 References

[N3886] Meneide, Working Draft

Working Draft

<https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3886.pdf>

[P0514R4] Giroux, Olivier

Efficient concurrent waiting for C++ 20

<https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0514r4.pdf>