# A Practical and Unexpected Problem Concerning Condition Variables

OR

C++ condition variables made easy
WTF?
Of course not!

(c) Thorsten Wendt
[email: thorsten.wendt@web.de]

29 April 2025

## Preliminary note

The intention of this talk was to explain a rather unknown, but nevertheless important quirk of C++ condition variables.

But after some discussions I hooked on the idea to give an introduction on std::mutex and std::condition_variable.

To bring these opposed topics under one hood though I've decided to omit a lot of details for the sake of clarity.

## What a std::mutex is good for (1/2)

Example: Numbers from *min_count* up to *max_count* will be added and accessing an unprotected shared resource **cnt**.

```
std::uint64_t cnt{0};    // <== Shared resource

void unsafe_add(std::uint64_t min_count, std::size_t max_count)
{
  for( std::uint64_t number = min_count; number <= max_count; ++number)
  {
    cnt = cnt + number;  // <== Accessing the shared resource
  }
}
```

If this functions is called by multiple threads the result will be unpredictable.

## What a std::mutex is good for (2/2)

In a multi threaded environment a *mutex* will ensure that only one thread will gain access to a shared resource.

```cpp
std::uint64_t cnt{0};               // <== Shared resource
std::mutex mutex;                   // <== Protects the shared resource

void add(std::uint64_t min_count, std::uint64_t max_count)
{
  std::lock_guard guard(mutex);     // <== Prevents concurrent access

  for( std::uint64_t number = min_count; number <= max_count; ++number)
  {
    cnt = cnt + number;
  }
}
```

## Using a std::mutex sanely.

### Let's start with a simple Queue

- Queues are widely used and can be find almost everywhere.
- Generally values can be added to a Queue and retrieved from it.
- The order is first in first out (FIFO).
- Queues are means to decouple threads and to avoid shared resources. (e.g Actor model, Apples GDC, . . . )

## Example using a Queue

```cpp
template<typename T>
class Queue
{
public:
  void push(T&& value)
  {
    std::lock_guard guard{mutex_};
    queue_.push(std::move(value));
  }

  std::optional<T> pop()
  {
    std::lock_guard guard{mutex_};
    if (queue_.empty())
    {
      return {};
    }
    std::optional<T> value{queue_.front()};
    queue_.pop();
    return value;
  }

private:
  std::queue<T> queue_;
  std::mutex mutex_;
}
```

## Here's how to use the Queue

```cpp
Queue<int> queue;

queue.push(1);                // Queue contains one value
queue.push(2);                // Queue now contains two values

auto value = queue.pop();  // *value == 1, Queue contains one value
value = queue.pop();         // *value == 2, Queue is empty
value = queue.pop();         // value is "empty"
```

But wait! How to wait for the next value?

# How to wait for the next value?

## Option I: Loop for a value

```
Queue<int> queue;
std::optional<int> value;

do
{
  value = queue.pop();
} while (!value);

do_something_with(value);
```

## How to wait for the next value?

### Option II: Move loop into the queue

```
T pop() noexcept       // <== No optional anymore
{
  for(;;) {            // <== endless loop
    mutex_.lock();
    if (!queue_.empty())
      break;
    mutex_.unlock();  // <== to give a thread a chance to push a value
  }
  T value{queue_.front()};
  queue.pop();
  mutex_.unlock();
  return value;
}
```

## How to wait for the next value?

Option II is better, because

- We got rid of the std::optional
- A call to Queue::pop() will always return a value.
- Cleaner interface

**But**

- We are burning CPU cycles if Queue is empty
  - We could introduce a sleep but how long we need to sleep? nano seconds, milli seconds, seconds?
- For the *push* thread it's hard to squeeze in between the "*pop loop*"

## How to wait for the next value?

### Option III: What if we get informed when a new value arrives?

```
T pop() noexcept
{
  mutex_.lock();
  if (queue_.empty()) {
    wait_for_new_value(mutex);  // <== Going to sleep and wait for new value
  }                             // <== Mutex is locked
  T value{queue_.front()};
  queue.pop();
  mutex.unlock();
  return value;
}
```

## How to wait for the next value?

This is cool! This would solve all our problems!

- No CPU cycle burning any more
- Fast reaction on arriving of new values
- Clean interface

**But**: How to do it?

## A condition variable can be used to signal new values

**[Wikipedia]**

Conceptually, a condition variable is . . . , associated with a mutex, on which a thread may wait for some condition to become true.

**[en.cppreference.org]**

The condition variable is a synchronization primitive used with a std::mutex to block one or more threads until another thread **both** modifies a shared variable (the condition) and notifies the condition variable.

## Using a condition variable (1/2)

### Waiting for a signal

```
std::condition_variable::wait(...)

template< class Predicate >
void wait( std::unique_lock<std::mutex>& lock, Predicate stop_waiting );
```

### [en.cppreference.org]

*wait* causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied . . .

## Using a condition variable (2/2)

**Important:**

- *spurious wakeup* means a thread may wakeup even if the thread wasn't notified.

**This means**

- Always check the condition (predicate) before keep going on **!!!**

## Handle the notification when a new value emerges

```
T pop()
{
  std::lock_guard guard{mutex_};
  while (queue_.empty()) {      // <== Check condition (queue is empty)
    cond_.wait(guard);          // <== Unlock mutex and Wait for the signal
  }
  T value{queue_.front()};      // <== Mutex is locked
  queue_.pop();
  return value;
}
...
std::condition_variable cond_;
```

Before the thread is going to wait for the signal the condition will be checked

- If the queue is empty the mutex will be released and the thread is going to sleep until it get notified/signaled
- If the queue is NOT empty we continue and don't fall into sleep. The mutex is kept locked.
- If spurious wakeup happens the loop will check the condition

# Loopless version on how to handle notification

```
T pop()
{
  std::lock_guard guard{mutex_};
  cond_.wait(guard, !queue.empty()); // <== Wait and check (1)
  T value{queue_.front()};
  queue_.pop();
  return value;
}
...
std::condition_variable cond_;
```

Before the thread is going to sleep (1) the predicate will be checked

- If the queue is empty the mutex will be released and the thread is going to sleep until it get notified/signaled
- If the queue is NOT empty we continue and don't fall into sleep. The mutex is kept locked.
- If spurious wake happen it will be checked/handled internally

# Signalling (notify_one)

## Using notify_one to signal a waiting thread

```
std::condition_variable::notify_one

void notify_one() noexcept;
```

## [en.cppreference.org]

If any threads are waiting on *this, calling notify_one unblocks one of the waiting threads.

. . .

The notifying thread does not need to hold the lock on the same mutex as the one held by the waiting thread(s); in fact doing so is a pessimization, since the notified thread would immediately block again, waiting for the notifying thread to release the lock.

## How to notifiy a new value will be emerged in the queue?

```
void push(T&& value)
{
  {
    std::lock_guard guard{mutex_};
    queue_.push(std::move(value));
  }
  cond_.notify_one();
}
```

All this is great! Storing a new value into Queue will signal a waiting thread a new value is available

**But wait:** Is this the end of the story?

Let's do some measurements

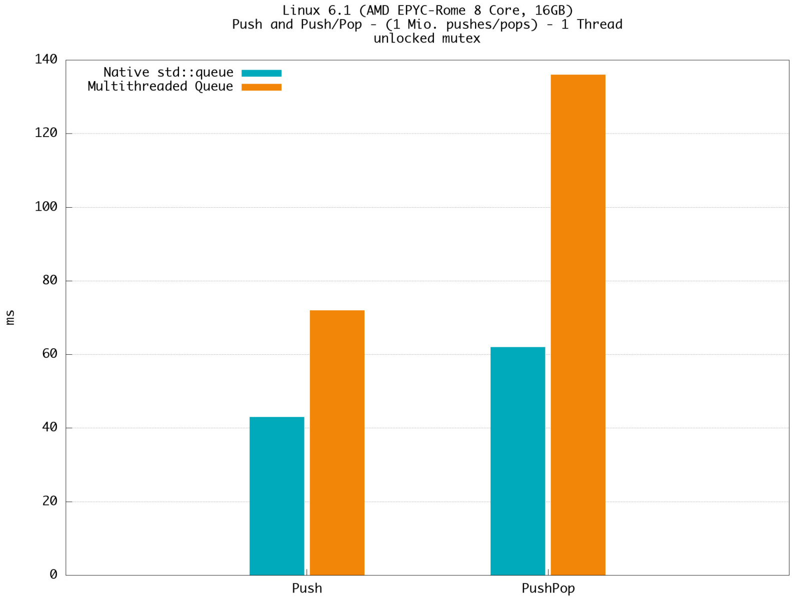


Figure 1: Pushing and popping 1 Mio values

## Let's do some assuptions

We are pushing in total 1.Mio ($10^6$) intergers into Queue and popping them out. On a 8 core Linux server we have the following four scenarios

1) **One reader** and **one writer** thread - SRSW
2) **10 reader** threads and **one writer** thread - MRSW
3) **One reader** thread and **10 writer** threads - SRMW
4) **10 reader** threads and **10 writer** threads - MRMW

- **What is the slowest and the fastest scenario?**
- **How much time does it take?**

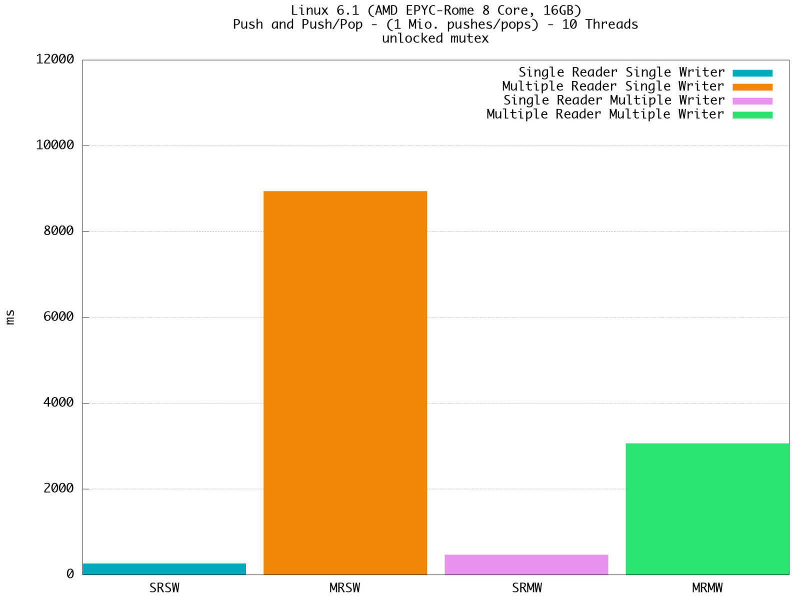


Figure 2: Pushing and popping 1 Mio values with 10 threads on Linux

## Wow - That knocks my socks off

## Let’s do some measurements on MacOS

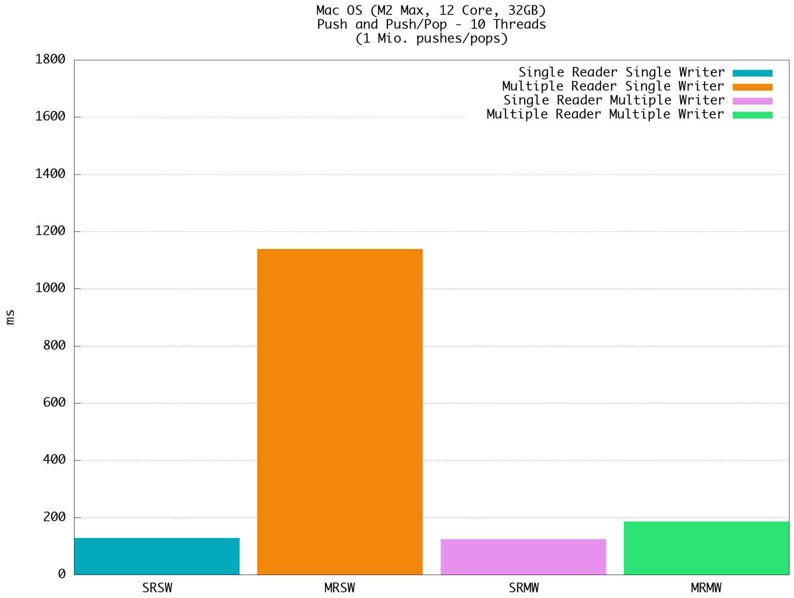


Figure 3: Pushing and popping 1 Mio values with 10 threads on MacOS

## Let’s do some measurements on Windows

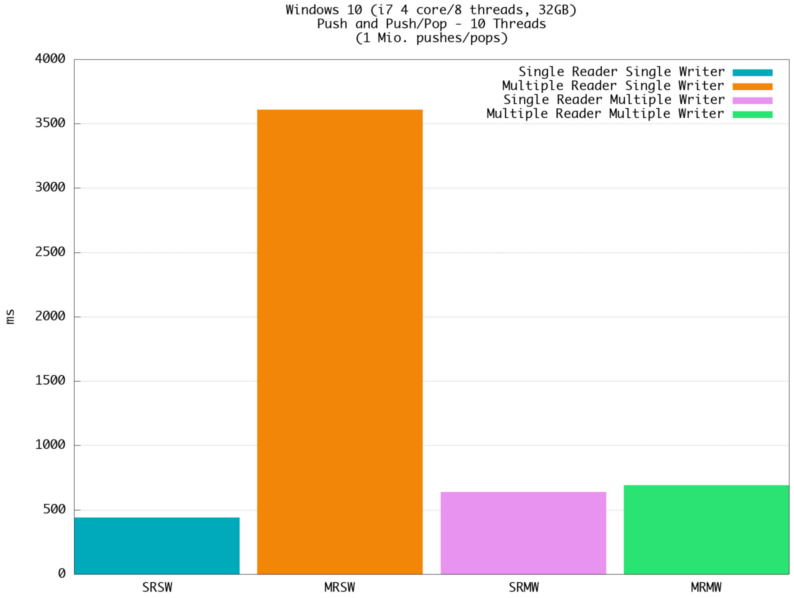


Figure 4: Pushing and popping 1 Mio values with 10 threads on Windows

## Let's read the documentation again

### [en.cppreference.org]

The notifying thread does not need to hold the lock on the same mutex as the one held by the waiting thread(s); in fact doing so is a pessimization, since the notified thread would immediately block again, waiting for the notifying thread to release the lock. However, some implementations (in particular many implementations of pthreads) recognize this situation and avoid this "hurry up and wait" scenario by transferring the waiting thread from the condition variable's queue directly to the queue of the mutex within the notify call, without waking it up.

Notifying while under the lock may nevertheless be necessary when precise scheduling of events is required, e.g. if the waiting thread would exit the program if the condition is satisfied, causing destruction of the notifying thread's condition variable. A spurious wakeup after mutex unlock but before notify would result in notify called on a destroyed object.

## Rewrite our Queue and remove the brackets

```
void push(T&& value)
{
  {                                       // <== Remove this bracket
    std::lock_guard guard{mutex_};
    queue_.push(std::move(value));
  }                                       // <== Remove this bracket
  cond_.notify_one();
}
```

Remove the curley braces to hold the lock while notifiying a thread. Then our queue code looks like below

```
void push(T&& value)
{
  std::lock_guard guard{mutex_};
  queue_.push(std::move(value));
  cond_.notify_one();
}
```

Let’s do some measurements on Linux again

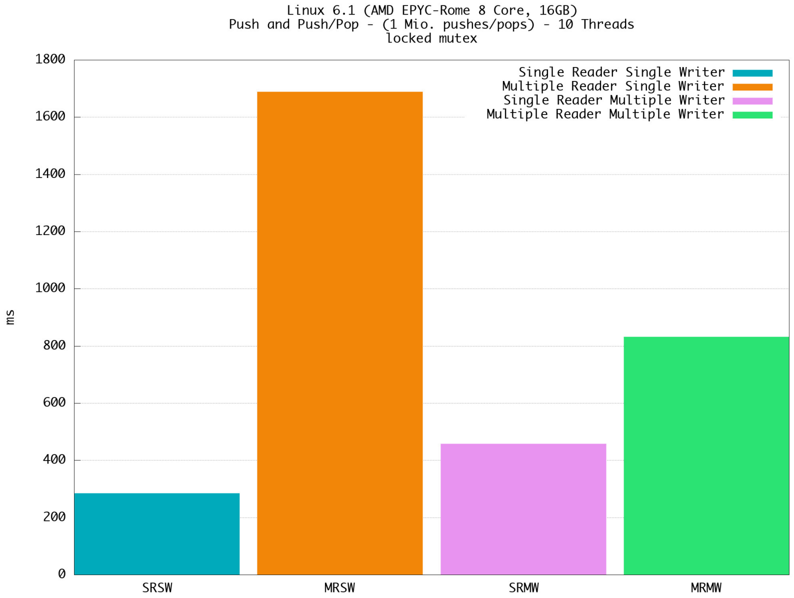
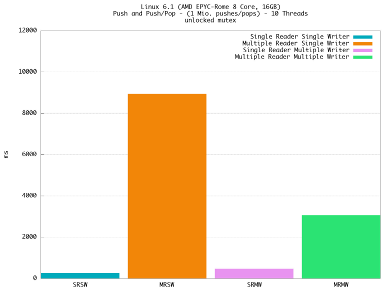


Figure 5: Pushing and popping 1 Mio values with 10 threads

# Let's compare the results



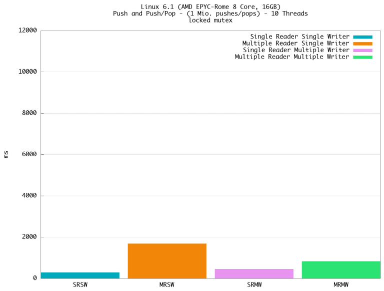


Figure 6: Comparing unlocked and locked version of Queue

## Recommendation

On Mac and on Windows there is no difference in performance if the mutex is locked or not while signalling a waiting thread.

Always lock the corresponding mutex while signalling a waiting thread !!!

## Lessons learned

Test, Test, Test and look at the numbers

Read The F. . . . . . Manual carefully

## Thank you

I would like to thank all the people I worked with and I met - I learned so much from you!

I would like to thank all the people behind the following projects and web sites for their effort to bring all that up to life.

- DevDocs.io - https://devdocs.io
- Pandoc - https://pandoc.org
- Cpp Reference - https://en.cppreference.com
- Beamer theme matrix - https://hartwork.org/beamer-theme-matrix
- Wikipedia - https://en.wikipedia.org