vollmann engineering gmbh

>

Concurrent Queues API

P0260R13

LEWG

December, 2024

Detlef Vollmann

vollmann engineering gmbh

Concurrent Queues are not Containers

- Concurrent queues are concurrent data structures
- A communication mechanism
- A synchronization mechanism
  - consumers wait for producers
  - producers wait for consumers
- (Temporary) storage is a possible implementation detail
  - queues with size 0 sometimes make perfect sense

Design Space

- The design space for concurrent queues is pretty big
  - partly in interface design
  - more in semantics
- Single or multiple connections on producer and/or consumer side
- Lock-free vs. locking
  - separate for both ends
- Memory allocation
  - up-front, per push/pop, external
- Ordering guarantee
  - FIFO vs. priorities
- Non-blocking only vs. `wait_*` interface
- Single push/pop vs. two-phase
- Strongly typed vs. (dynamically sized) byte chunks

Design Space

- More interface
  - timed waits
  - asynchronous
  - debugging
  - single ended interfaces
- Efficiency vs. robust/portable interface
- Error handling (exceptions)
- Concurrency vs. parallelism vs. asynchronicity

Concepts for Concurrent Queues

- No single queue implementation can cover all design aspects
- Provided concepts are expected to cover most design aspects
- Implementing both async and non-blocking interfaces has performance costs
- Concept is split into one base concept and two separate concepts for async and non-blocking
- Many different implementations for these concepts are expected
  - some of them may be standardized
- Possible single-ended adapter can use these concepts
- `bounded_queue` models all concepts

”Error” Handling

- ”One person's exception is another person's expected result.”
- The current proposal is to have no queue based errors.
- LEWG decided in Wroclaw to have `optional<T> pop()`
  - i.e. `closed` is not an error
- This leads to `bool push(T&& x)`
- For non-blocking functions (`try_*`) `empty` and `full` (and arguably `busy`) are similar
- Now `conqueue_error` and `conqueue_category` are not needed anymore and `conqueue_errc` should possibly renamed (was `queue_op_status` before R5).

Closing Queues

- The only queues that don't need `close` are
  - queues that are never closed
  - single producer, single consumer with inline close token
- For all other cases synchronization needs access to queue internals
  - as detailed in the paper
- So the basic concept contains `close`

Example

- ”Find files with string”
- One task/thread collects all the file paths in a directory and pushes them into a queue and then closes the queue
- Other tasks/threads (one or more) pop file paths from the queue and search them for a string
- Synchronous version with multiple threads
- Single-threaded Asynchronous version with coroutines
- Single-threaded Asynchronous version with native S/R
- Code available at
  `https://gitlab.com/cppzs/conqueue/-/tree/wg21-demos/demo`

Synchronous `push`

- Push interface

```
bool push(const T& x);
bool push(T&& x);
template <typename... Args> bool emplace(Args &&... as);
```

- Returns `true` on success and `false` on close

Synchronous `pop`

- Pop interface

  ```
  optional<T> pop();
  ```

- Returns `optional` with value on success and empty `optional` on close
- This was what LEWG voted for in Wroclaw

Non-Blocking push

- Push interface

```
conqueue_errc try_push(const T& x);
conqueue_errc try_push(T&& x);
template <typename... Args> conqueue_errc try_emplace(Args &&... as);
```

- This is the logical extension to blocking push

Non-Blocking pop

- Pop interface

  ```
  optional<T> try_pop(conqueue_errc &ec);
  ```

- Alternative versions would be

  ```
  expected<T, conqueue_errc> queue::try_pop();
  ```

- or even

  ```
  expected<optional<T>, conqueue_errc> queue::try_pop();
  ```

Non-Blocking `pop`

- Example from P2921R0:

```
conqueue_errc ec;
while (auto val = q.try_pop(ec))
   println(”got {}”, *val);
if (ec == conqueue_errc::closed)
  return;
// do something else.
```

- With `expected<T, conqueue_errc>`

```
auto val = q.try_pop();
while (val) {
  println(”got {}”, *val);
  val = q.try_pop();
}
if (val.error() == conqueue_errc::closed)
  return;
// do something else
```

Non-Blocking `pop`

- With `expected<optional<T>, conqueue_errc>`
  ```
  auto val = q.try_pop();
  while (val && *val) {
    println(”got {}”, **val);
    val = q.try_pop();
  }
  if (val.error() == conqueue_errc::closed)
    return;
  // do something else
  ```
- LEWG poll in St. Louis: ”LEWG would like to add a `std::expected` interface for concurrent queues”:
  `|SF|F|N|A|SA| |0|2|5|3|2"`

Asynchronous Interface

- Pop interface

  ```
  sender auto async_pop();
  ```

- LEWG voted strongly in favour in Wroclaw for the sender to call `set_value(optional<T>)`
- Sender/receiver are used via coroutines or native
- For coroutines, `set_value(optional<T>)` is probaly the perfect choice
- For native sender/receiver using two value channels is probably a much better choice
- Different interfaces for coroutines and native are akward
- Coroutines should provide additional infrastructure to make use of native interface more handy
  - e.g. `as_optional`

  ```
  while ((fname = co_await (files->async_pop() | as_optional())))
  ```

- Proposal: calls `set_value(T)` on success and `set_value()` when closed.

Asynchronous Interface

- Push interface

```
sender auto async_push(const T&); // sends void (success), conqueue_errc
sender auto async_push(T&&);
template <typename... Args> sender auto async_emplace(Args &&... as);
```

- Analogous to `async_pop` it calls `set_value(true_type)` on success and `set_value()` when closed.

Complete proposed API

```
allocator_type get_allocator() const noexcept;
void close() noexcept;
bool is_closed() const noexcept;

bool push(const T& x);
bool push(T&& x);
template <typename... Args> bool emplace(Args &&... as);
optional<T> pop();

conqueue_errc try_push(const T& x);
conqueue_errc try_push(T&& x);
template <typename... Args> conqueue_errc try_emplace(Args &&... as);
optional<T> try_pop(conqueue_errc &ec);

sender auto async_push(const T&); // sends true_type (success), void (closed)
sender auto async_push(T&&);
template <typename... Args> sender auto async_emplace(Args &&... as);
sender auto async_pop();  // sends T (success), void (closed)
```