Saks & Associates

Saks & Associates
393 Leander Drive
Springfield, OH 45504-4906 USA
+1-412-521-4117 (voice)
service@saksandassociates.com
saksandassociates.com

2

About Ben Saks

Ben Saks is the president of Saks & Associates. He is the principal editor and presenter for much of Saks & Associates' training curriculum on the use of C and C++ in embedded systems.

Ben has represented Saks & Associates on the ISO C++ Standards committee as well as two of the committee's study groups:

- SG14 — low-latency
- SG20 — education

Ben has spoken at industry conferences, including *CppCon: The C++ Conference*, the *C++ and System Software Summit*, the *Embedded Systems Conference*, and *NDC Techtown*. At *CppCon*, he's the chair of the Embedded Track and a member of the program committee.

3

More About Ben Saks

Ben previously worked as a software engineer for Vorne Industries, where he used C++ and JavaScript to develop embedded systems that help improve manufacturing productivity in factories all over the world. He is a contributing author on multiple Vorne patents.

Ben earned a B.A. with Distinction in Computer Science from Carleton College.

4

Outline

- Introduction
- An Example Using Atomics
- Non-Preemptibility
- Synchronizability
- Implementing `my_atomic`

5

Concurrency and Multithreading

- Many modern programs use some form of ***concurrency***, meaning that the program appears to be working on multiple tasks simultaneously.
- One common approach is to make the program ***multithreaded*** — that is, use multiple threads of execution.
- Some systems are ***hardware concurrent*** — they can literally run multiple threads simultaneously (e.g., by using multiple processor cores).
- Other systems simply make it appear as if tasks are running simultaneously by switching between them rapidly.

6

Inter-Thread Communication

- Multithreaded programs often need some way to communicate between threads.
- Atomic objects and operations are one tool for communicating between threads.
- Others include:
  - Mutexes
  - Semaphores
  - Condition variables

7

`std::atomic<T>`

- The standard header <*atomic*> provides the class template `std::atomic<T>` to ensure that a `T` object is accessed atomically.
- As of the C++23 standard, <*atomic*> is supposed to be available even in freestanding implementations.
- Unfortunately, not everyone has access to a C++23 toolchain that implements this aspect of the standard.

8

My Situation

- In my case, I was experimenting with multithreaded code for the Arduino MEGA 2560.
- I was trying to implement a queue using `std::atomic<T>` similar to what Charles Frasch presented at CppCon 2023. [Frasch (2023)].
- I was using the Arduino IDE 2.3.6 to build and deploy the code.
- Unfortunately, that environment didn't provide an <*atomic*> header containing the tools that I needed.
- As such, I implemented my own version of `std::atomic<T>`.

9

Defining "Atomic"

- Before we dig into that, let's discuss what makes an operation "atomic".
- When a programmer says that an operation is "atomic", there are at least two properties to which they might be referring:
  - ***Non-preemptible*** — The operation can't be preempted in the middle (e.g., by another thread)
  - ***Synchronizable*** — The results of the operation can be made visible to other threads of execution in a controllable fashion
- For this discussion, an ***atomic object*** is simply an object that supports a minimal set of atomic operations.
- We'll look at how to achieve these properties after a brief overview of how the queue works…

10

Outline

- Introduction
- An Example Using Atomics
- Non-Preemptibility
- Synchronizability
- Implementing `my_atomic`

11

How the Queue Works

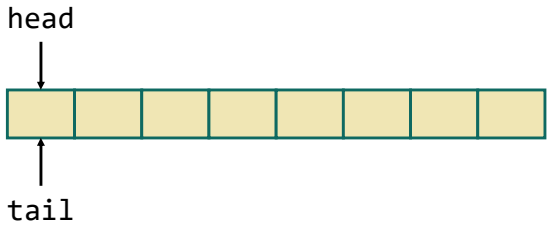
- The queue design that I used works only for single-producer, single-consumer applications.
- I used a queue of 32-bit integers, but you could adapt the design to support other data types.
- The elements are stored in a fixed-size array:

- This data structure is sometimes called a “circular buffer” or “ring buffer”, so I chose `ring_buffer` for its name.

12

How the Queue Works

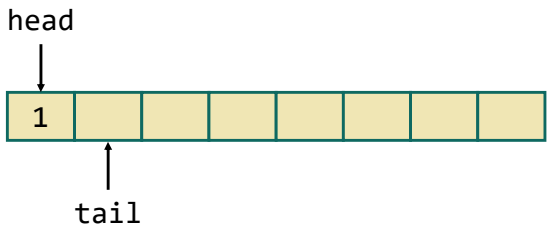
- In addition to the array, `ring_buffer` contains two indices into the array:
  - `head` — The index of the "front" element, if any.
  - `tail` — The index where a new "back" element would be inserted.
- In an initially-empty container, both `head` and `tail` start at zero:



13

How the Queue Works

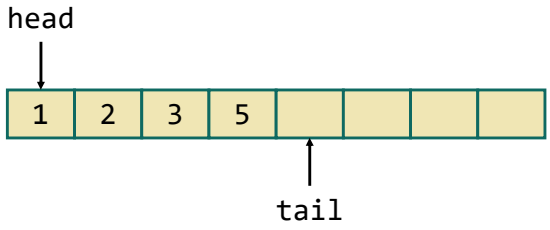
- `ring_buffer::try_push_back` attempts to append a new element to the queue.
- If the queue isn't full, it:
  - writes the new element to the tail location, and
  - advances `tail`



14

How the Queue Works

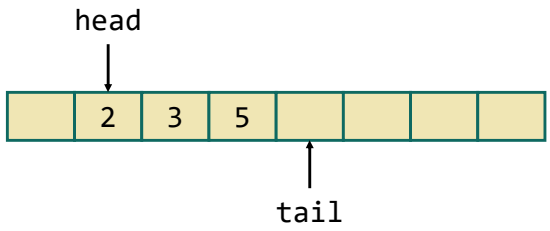
- Here's the queue with a few more elements added:



15

How the Queue Works

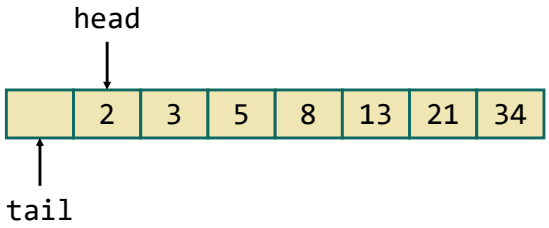
- `ring_buffer::try_pop` attempts to read and remove the element at the front of the queue.
- If the queue isn't empty, it:
  - advances `head`, and
  - gives back the element that was at the front.



16

How the Queue Works

- Normally, advancing an index is a simple increment.
- If the increment would place the index out of bounds, it wraps around to zero (that's what makes it a "ring").



17

How the Queue Works

- As we'll see, it turns out that the indices `head` and `tail` need to be atomic objects.
- I wanted to be able to test my atomic class template with types of different sizes.
- Thus, I wrote `ring_buffer` in terms of a type alias `index_type`:

```
using index_type = uint8_t;


template <index_type N>
class ring_buffer { ~~~ };
```

18

How the Queue Works

- As we'll see, it turns out that the indices `head` and `tail` need to be atomic objects.
- I wanted to be able to test my atomic class template with types of different sizes.
- Thus, I wrote `ring_buffer` in terms of a type alias `index_type`:

```
using index_type = uint8_t;
                   uint16_t

template <index_type N>
class ring_buffer { ~~~ };
```

19

How the Queue Works

- Again, the storage for a `ring_buffer` consists of an array for the elements and indices for `head` and `tail`:

```
template <index_type N>
class ring_buffer {
public:
    using value_type = unsigned long;
    ~~~
private:
    static constexpr index_type size = N + 1;
    index_type head, tail;
    value_type buffer[size];
};
```

- It uses a built-in array because this toolchain doesn't provide *<array>*.

20

How the Queue Works

- Again, we'll eventually see that `head` and `tail` need to be atomic objects:

```
template <index_type N>
class ring_buffer {
public:
    using value_type = unsigned long;
    ~~~
private:
    static constexpr index_type size = N + 1;
    my_atomic<index_type> head, tail;
    value_type buffer[size];
};
```

- But for now, we'll try to do without the `my_atomic` template.

21

How the Queue Works

- The `size` constant adjusts for the fact that with this design, there's always at least one unused element in the array:

```
template <index_type N>
class ring_buffer {
public:
    using value_type = unsigned long;
    ~~~
private:
    static constexpr index_type size = N + 1;
    index_type head, tail;
    value_type buffer[size];
};
```

22

How the Queue Works

- Here's the class interface:

```
template <index_type N>
class ring_buffer {
public:
    using value_type = unsigned long;
    ring_buffer();
    bool empty() const;
    bool try_pop(value_type &v);
    bool try_push_back(value_type c);
private:
    ~~~
};
```

23

How the Queue Works

- There are a lot of ways that you could improve this code using qualifiers like:
  - `constexpr`
  - `[[nodiscard]]`
  - `noexcept`
- I've omitted those here to make it easier to see the important details.
- We'll look at how these member functions are implemented as we discuss the properties of atomic operations.

24

Outline

- Introduction
- An Example Using Atomics
- Non-Preemptibility
- Synchronizability
- Implementing `my_atomic`

25

`try_pop`, Take 1

- Here's a first attempt at implementing `try_pop`:

```
template <index_type N>
bool ring_buffer<N>::try_pop(value_type &v) {
    if (!empty()) {
        v = buffer[head];
        if (++head >= size) {
            head = 0;
        }
        return true;
    } else {
        return false;
    }
}
```

26

try_pop, Take 1

- Unfortunately, this section of code for advancing head has a couple of problems:

```
template <index_type N>
bool ring_buffer<N>::try_pop(value_type &v) {
    if (!empty()) {
        v = buffer[head];
        if (++head >= size) {
            head = 0;
        }
        return true;
    } else {
        return false;
    }
}
```

27

try_pop, Take 1

- The first problem occurs if this code is preempted while incrementing head:

```
template <index_type N>
bool ring_buffer<N>::try_pop(value_type &v) {
    if (!empty()) {
        v = buffer[head];
        if (++head >= size) {
            head = 0;
        }
        return true;
    } else {
        return false;
    }
}
```

28

Meanwhile…

- The `try_push_back` function (shown in part) also reads from `head`:

```
template <index_type N>
bool ring_buffer<N>::try_push_back(value_type v) {
    index_type next_tail {tail + 1};
    if (next_tail >= size) {
        next_tail = 0;
    }
    if (next_tail != head) {
        ~~~
    }
}
```

29

Meanwhile…

- `try_push_back` is always called by the producer thread, while `try_pop` is always called by the consumer thread.
- That's why `ring_buffer`'s design depends on:
  - reading and writing the indices non-preemptively, and
  - having only one producer thread and one consumer thread.
- If the consumer thread is preempted while writing `head`, the producer thread may read a half-written nonsense value, and vice versa.

30

Non-Preemptibility

- Some operations are ***inherently non-preemptible*** simply because they have just a single step.
- For example, an operation that takes only one processor cycle to complete is inherently non-preemptible.
- You can prevent a longer operation from being preempted by using a tool such as a lock.
- A programmer might describe using a lock to prevent preemption as "making an operation atomic".

31

Atomic Indices

- You can prevent this first problem by making `head` and `tail` into atomic objects:

```
template <index_type N>
class ring_buffer {
public:
    using value_type = unsigned long;
    ~~~
private:
    static constexpr index_type size = N + 1;
    my_atomic<index_type> head, tail;
    value_type buffer[size];
};
```

32

The my_atomic Class Template

- Here's a minimal version of the my_atomic class template definition:

```
template <typename T>
class my_atomic {
public:
    my_atomic() = default;
    my_atomic(T v);
    my_atomic(my_atomic const &other) = delete;
    my_atomic &operator=(my_atomic const &other) = delete;
    void store(T v);
    T load() const;
private:
    T value;
};
```

33

The my_atomic Class Template

- Notice that these functions use pass-by-value rather than pass-by-reference.

```
my_atomic(T v);      // not my_atomic(T const &v)
void store(T v);     // not void store(T const &v)
T load() const;      // not T const &load() const
```

- This is how the corresponding functions in std::atomic<T> operate, so I chose to mimic that here.

34

The my_atomic Class Template

- The constructors hold no surprises:

```
my_atomic() = default;
my_atomic(T v);
```

- To keep things simple, this version of `my_atomic` is neither copyable nor movable in the traditional sense:

```
my_atomic(my_atomic const &other) = delete;
my_atomic &operator=(my_atomic const &other) = delete;
```

- Instead, you must use `load` to read the underlying value and copy it.

35

The my_atomic Class Template

- `store` atomically sets a new value for the object:

```
void store(T v);
```

- `load` atomically reads out the object's value:

```
T load() const;
```

36

Atomic Objects

- None of the individual operations that `std::atomic<T>` supports is preemptible.
- These operations include:

```
std::atomic<int> x;
int y, z;
x.store(10);            // Set x to 10
y = x.load();           // Read the value of x into y
```

- If these operations aren't inherently non-preemptible, `std::atomic<T>` must employ a locking mechanism to prevent preemption.
- `my_atomic<T>` must reproduce this behavior.

37

Is It Lock-Free?

- By the way, you can determine if a specialization of `std::atomic<T>` uses a locking mechanism by examining the `is_always_lock_free` data member:

```
static_assert(std::atomic<std::int32_t>::is_always_lock_free,
    "Requires a lock-free 32-bit atomic type");
```

- If you need a lock-free atomic, this is normally the best way to check.
- That said, some `std::atomic<T>` specializations use locks conditionally (e.g., only if they're unaligned).
  - You can check if a specific object uses locks by calling `is_lock_free()`.

38

Updating an Index Non-Preemptively

- At first, you might imagine that the revised `try_push_back`'s code for updating `tail` would look something like this:

```
bool ring_buffer<N>::try_push_back(value_type v) {
    ~~~
    if (++tail >= size) {
        tail = 0;
    }
    ~~~
}
```

- Unfortunately, this still doesn't work because it updates `tail` in multiple steps…

39

Updating an Index Non-Preemptively

- While individual operations on an atomic object aren't preemptible, this code could be preempted between the two lines:

```
bool ring_buffer<N>::try_push_back(value_type v) {
    ~~~
    if (++tail >= size) {
        tail = 0
    };                    preempted here
    ~~~
}
```

- If the consumer thread preempts the code here, it could be confused because `tail` should never be equal to `size`.

40

Updating an Index Non-Preemptively

- Instead, `ring_buffer` must calculate the next value for `tail` in a temporary, then do the assignment non-preemptibly:

```
bool ring_buffer<N>::try_push_back(value_type v) {
    ~~~
    index_type next_tail {tail.load()};
    if (++next_tail >= size) {
        next_tail = 0;
    }
    tail.store(next_tail);
    ~~~
}
```

41

Updating an Index Non-Preemptively

- Since the code is very similar for updating `head`, `ring_buffer` actually uses the private member function `next` to calculate the next value for an index:

```
template <index_type N>
class ring_buffer {
public:
    ~~~
private:
    static index_type next(my_atomic<index_type> const &index);
    ~~~
};
```

42

Updating an Index Non-Preemptively

- Here's the implementation for the next function:

```
template <index_type N>
index_type ring_buffer<N>::next(my_atomic<index_type> const &index) {
    index_type next_index {index.load()};
    if (++next_index >= size) {
        next_index = 0;
    }
    return next_index;
}
```

43

try_push_back

- Here is the full implementation for try_push_back using next:

```
template <index_type N>
bool ring_buffer<N>::try_push_back(value_type v) {
    index_type next_tail {next(tail)};
    if (next_tail != head.load()) {
        buffer[tail.load()] = v;
        tail.store(next_tail);
        return true;
    } else {
        return false;
    }
}
```

44

try_push_back

- Here is the full implementation for try_push_back using next:

```
template <index_type N>
bool ring_buffer<N>::try_push_back(value_type v) {
    index_type next_tail {next(tail)};
    if (next_tail != head.load()) {  ← check if the queue is full
        buffer[tail.load()] = v;
        tail.store(next_tail);
        return true;
    } else {
        return false;
    }
}
```

45

try_push_back

- Here is the full implementation for try_push_back using next:

```
template <index_type N>
bool ring_buffer<N>::try_push_back(value_type v) {
    index_type next_tail {next(tail)};
    if (next_tail != head.load()) {
        buffer[tail.load()] = v;  ← store new element (preemptible)
        tail.store(next_tail);
        return true;
    } else {
        return false;
    }
}
```

46

try_push_back

- Here is the full implementation for try_push_back using next:

```
template <index_type N>
bool ring_buffer<N>::try_push_back(value_type v) {
    index_type next_tail {next(tail)};
    if (next_tail != head.load()) {
        buffer[tail.load()] = v;
        tail.store(next_tail);   <-- update tail (non-preemptible)
        return true;
    } else {
        return false;
    }
}
```

47

try_push_back

- This version of try_push_back loads from tail twice (once inside next):

```
template <index_type N>
bool ring_buffer<N>::try_push_back(value_type v) {
    index_type next_tail {next(tail)};
    if (next_tail != head.load()) {
        buffer[tail.load()] = v;
        tail.store(next_tail);
        return true;
    } else {
        return false;
    }
}
```

- Here's a faster but less readable version that avoids this issue…

48

```
template <index_type N>
bool ring_buffer<N>::try_push_back(value_type v) {
    index_type current_tail {tail.load()};
    // calculate next_tail
    index_type next_tail {current_tail + 1};
    if (next_tail == size) {
        next_tail = 0;
    }
    if (next_tail != head.load()) {
        buffer[current_tail] = v;
        tail.store(next_tail);
        return true;
    } else {
        return false;
    }
}
```

49

try_pop, Take 2

- In try_pop, updating head non-preemptibly is similarly important:

```
template <index_type N>
bool ring_buffer<N>::try_pop(value_type &v) {
    if (!empty()) {
        v = buffer[head.load()];        // preemptible read into v
        head.store(next(head));         // non-preemptible write
        return true;
    } else {
        return false;
    }
}
```

50

empty

- You might reasonably be concerned that `empty` could be preempted between `head.load()` and `tail.load()`:

```
template <index_type N>
bool ring_buffer<N>::empty() {
    return head.load() == tail.load();
}
```

- While preemption is possible here, it doesn't cause any problems here.
- `empty` is called only from the consumer thread.
- `tail` would change only if the producer thread interrupts to make the queue non-empty.

51

Preemptible Element Access

- While `ring_buffer` must update its `head` and `tail` indices non-preemptively, it doesn't require any additional locking mechanisms.
- Only the producer thread calls `try_push_back`.
  - `try_push_back` writes the new element before updating `tail`.
  - Preemption is okay while writing the new element, because it's invisible to the consumer until `tail` advances.
- Only the consumer thread calls `try_pop`.
  - `try_pop` updates `head` only after reading the front value.
  - Again, preemption is okay while reading the front value, because `try_push_back` will refuse to overwrite it until `head` advances.

52

Outline

- Introduction
- An Example Using Atomics
- Non-Preemptibility
- Synchronizability
- Implementing `my_atomic`

53

Synchronization

- As previously mentioned, only the accesses to the indices `head` and `tail` need to be non-preemptible.
- However, it's also important that the indices advance *after* the element is read/written.
- This brings up the problem of synchronization…

54

Synchronization

- In general, the results of an operation completed on some thread A aren't guaranteed to be immediately visible to another thread B.
- For example, suppose that thread A is running on processor core #1 and thread B is running on a different processor core #2.
- Cores #1 and #2 may have separate caches.
- Core #2's cache may not always reflect the most recent changes made by operations on core #1 (and vice versa).
- Modern computers and optimization techniques are complex — caching is just one of the ways for a synchronization problem to arise.

55

Synchronization

- More generally, the standard says that operations on thread A and thread B are ***unsequenced*** if they don't ***synchronize with*** each other.
- Without synchronization, thread A and thread B might see the same events happen in two different orders.

56

Synchronization

- Consider the following code:

```
int x {0}, y {0};

void threadA() {
    x = 1;          // #1
    y = 1;          // #2
}

void threadB() {
    int i = x;
    int j = y;
    ~~~
}
```

- On the surface, `threadA` appears to write to `x` before writing to `y`.
- As such, it shouldn't be possible for `threadB` to read `x` as `0` and `y` as `1`.
- However…

57

Synchronization

- Consider the following code:

```
int x {0}, y {0};

void threadA() {
    y = 1;          // was #2
    x = 1;          // was #1
}

void threadB() {
    int i = x;
    int j = y;
    ~~~
}
```

- The optimizer might choose to reorder the lines.
- After all, from the compiler's perspective, `x` and `y` are unrelated.

58

Synchronization

- Consider the following code:

```
int x {0}, y {0};

void threadA() {
    x = 1;          // #1
    y = 1;          // #2
}

void threadB() {
    int i = x;
    int j = y;
    ~~~
}
```

- Even with the original ordering, there's no guarantee that `threadB` will see the results of #1 before #2.
- For example, if `x` and `y` are on different cache lines, `y`'s cache line might be written to memory before `x`'s cache line.
- Consider what this means for `ring_buffer`…

59

Synchronization in `try_pop`

- Here again is the code for `try_pop`:

```
template <index_type N>
bool ring_buffer<N>::try_pop(value_type &v) {
    if (!empty()) {
        v = buffer[head.load()];        // #1
        head.store(next(head));         // #2
        return true;
    } else {
        return false;
    }
}
```

60

Synchronization in try_pop

- If the producer thread sees line #2 happen *before* line #1, it could allow try_push_back to overwrite the front element before #1 completes:

```
template <index_type N>
bool ring_buffer<N>::try_pop(value_type &v) {
    if (!empty()) {
        v = buffer[head.load()];          // #1
        head.store(next(head));           // #2
        return true;
    } else {
        return false;
    }
}
```

- This code is safe only if #2 is a "synchronization operation"…

61

Synchronization Operations

- The C++ standard classifies these operations as ***synchronization operations***:
  - Operations on mutexes (and similar tools like semaphores)
  - Many (but not all) operations on atomic objects
- The producer thread can synchronize with the consumer thread only if each thread contains a synchronization operation.

62

Synchronization Operations

- In broad terms, there are two kinds of synchronization operations for atomic objects:
  - ***release operations***, which are typically write operations, and
  - ***acquire operations***, which are typically read operations.
- Thread A ***synchronizes with*** thread B if:
  - thread A and thread B both access a shared object X,
  - thread A writes to X (i.e., performs a release operation on X), and
  - thread B reads from X (i.e., performs an acquire operation on X) and receives the value written by thread A.

63

Synchronization Operations

- By default, every operation on a `std::atomic<T>` object is both a release operation and an acquire operation.
- Thus, we'll focus on how to make an atomic operation both a release operation and an acquire operation.

64

Outline

- Introduction
- An Example Using Atomics
- Non-Preemptibility
- Synchronizability
- Implementing `my_atomic`

65

The `my_atomic` Class Template

- How can you implement this minimal `my_atomic` class template?

```
template <typename T>
class my_atomic {
public:
    my_atomic() = default;
    my_atomic(T v);
    my_atomic(my_atomic const &other) = delete;
    my_atomic &operator=(my_atomic const &other) = delete;
    void store(T v);
    T load() const;
private:
    T value;
};
```

66

The `my_atomic` Class Template

- `my_atomic`'s non-trivial constructor simply calls `store`:

```
template <typename T>
my_atomic<T>::my_atomic(T v) {
    store(v);
}
```

67

Disabling Interrupts

- Again, some multithreading systems are not hardware concurrent — they can't literally execute two threads simultaneously.
- Many multithreading systems have a ***scheduler*** that decides how and when to execute each thread.
- On a system that isn't hardware concurrent, preventing the scheduler from running effectively prevents preemption.
- In most such systems, the scheduler is controlled by an interrupt.
- Thus, disabling interrupts prevents preemption on these systems.
- This is the technique that the Arduino MEGA 2560 datasheet uses.

68

Disabling Interrupts

- Disabling interrupts is usually a platform-specific operation.
- The Arduino MEGA 2560 has a Global Interrupt Enable bit in the status register SREG.
- You can write to SREG directly, or you can use the `sei()` and `cli()` macros, respectively, to set and clear just that one bit.

69

Disabling Interrupts

- The Arduino MEGA 2560 datasheet suggests that you write code to disable interrupts something like this:

```
uint16_t volatile &hw_reg {~~~};

uint16_t next_reg_val {~~~};
char cSREG;
cSREG = SREG;           // record the existing value of SREG
cli();                  // disable interrupts
hw_reg = next_reg_val;  // protected write
SREG = cSREG;           // restore the original value of SREG
```

70

Disabling Interrupts

- In C++, you should generally create an RAII type like this one for disabling interrupts:

```
class interrupt_disabler {
public:
    interrupt_disabler(): SREG_image (SREG) {
        cli();
    }
    ~interrupt_disabler() {
        SREG = SREG_image;
    }
private:
    uint8_t SREG_image;
};
```

71

Disabling Interrupts

- Here's a version of `my_atomic<T>::load` that uses this `interrupt_disabler` class:

```
template <typename T>
T my_atomic<T>::load() const {
    T rv;
    {
        interrupt_disabler disable_interrupts;
        rv = value;
    }
    return rv;
}
```

72

Disabling Interrupts

- And here's a corresponding version of `my_atomic<T>::store` that uses the `interrupt_disabler` class:

```
template <typename T>
void my_atomic<T>::store(T v) {
    interrupt_disabler disable_interrupts;
    value = v;
}
```

73

Disabling Interrupts and Synchronization

- You've seen how disabling interrupts prevents preemption, but how does it synchronize with other operations?
- Here's how the `sei()` and `cli()` macros are implemented in *<avr/interrupt.h>* when compiling with GCC:

```
#define sei()  __asm__ __volatile__ ("sei" ::: "memory")
#define cli()  __asm__ __volatile__ ("cli" ::: "memory")
```

- What are those expressions doing?

74

Disabling Interrupts and Synchronization

```
#define sei()  __asm__ __volatile__ ("sei" ::: "memory")
#define cli()  __asm__ __volatile__ ("cli" ::: "memory")
```

- `__asm__` is a GNU extension that allows you to embed assembly instructions in C/C++ code.

75

Disabling Interrupts and Synchronization

```
#define sei()  __asm__ __volatile__ ("sei" ::: "memory")
#define cli()  __asm__ __volatile__ ("cli" ::: "memory")
```

- `__volatile__` is a qualifier on `__asm__`.
- It tells the compiler that this assembly code has side effects, meaning that the compiler must not perform certain optimizations.
- Without `__volatile__`, the compiler would assume that the assembly code affects only the input and output values (these blocks have none of either).

76

Disabling Interrupts and Synchronization

```
#define sei()  __asm__ __volatile__ ("sei" ::: "memory")
#define cli()  __asm__ __volatile__ ("cli" ::: "memory")
```

- `sei` and `cli` are AVR assembly instructions (the Arduino MEGA 2560 uses the AVR instruction set).
- `sei` sets the Global Interrupt Enable flag.
- `cli` clears that flag.

77

Disabling Interrupts and Synchronization

```
#define sei()  __asm__ __volatile__ ("sei" :: "memory")
#define cli()  __asm__ __volatile__ ("cli" :: "memory")
```

- If either assembly block had output operands, they would be listed after the first `:`, marked in red.
- Any input operands would be listed after the second `:`, marked in teal.

78

Disabling Interrupts and Synchronization

```
#define sei()  __asm__ __volatile__ ("sei" ::: "memory")
#define cli()  __asm__ __volatile__ ("cli" ::: "memory")
```

- The third : begins the "clobbers" section, which must list the registers that are overwritten by the assembly block.
- In the subsequent code, the compiler must assume that any values that were stored in the listed registers before the block are gone.
- The special clobbers argument "memory" is the critical part for synchronization…

79

Disabling Interrupts and Synchronization

- The special clobbers argument "memory" says (more or less) that the assembly block affects all of memory.
- As such, the compiler must generate code to:
  - flush any writes currently held in registers and cache to memory before executing the assembly block, and
  - reload any values held in registers or cache from memory after the block.
- This is one way to synchronize with other threads.
- In essence, "memory" establishes what's sometimes called a ***memory barrier***.

80

Direct Assembly

- You can also use this technique to employ assembly instructions that perform atomic operations directly.
- This block uses x86 assembly to increment `x` atomically:

```
int x;
__asm__ __volatile__("lock xaddl %1, %0"
    : "+m"(x)        // %0: memory operand, read/write
    : "r"(1)         // %1: literal 1
    : "memory");
```

- This works even on a multicore system — the `lock` qualifier prevents other cores from interfering with the operation.

81

Compiler Intrinsics

- Alternatively, your compiler may provide you with tools for performing atomic operations.
- For example, GCC may offer these built-in functions:
  - `__atomic_load_n`
  - `__atomic_store_n`
- These functions (if present) should work for both integral and pointer types.

82

Compiler Intrinsics

- The first argument to each of GCC's intrinsic `__atomic_*` functions is a pointer to the atomic object.
- This call reads a 2-byte value from `atomic_obj` into `x`:

```
uint16_t *atomic_obj = ~~~;
uint16_t x = __atomic_load_n(atomic_obj, /* op type */);
```

- This call stores a 4-byte value from `val` to `atomic_obj2`:

```
uint32_t *atomic_obj2 = ~~~;
uint32_t val = 100;
__atomic_store_n(atomic_obj2, val, /* op type */);
```

83

Compiler Intrinsics

- The final argument to these `__atomic_*` functions is effectively an enumeration value that corresponds to the `std::memory_order` enumeration:
  - `__ATOMIC_RELAXED`
  - `__ATOMIC_CONSUME`
  - `__ATOMIC_ACQUIRE`
  - `__ATOMIC_RELEASE`
  - `__ATOMIC_ACQ_REL`
  - `__ATOMIC_SEQ_CST`
- `__ATOMIC_SEQ_CST` is the equivalent of `std::atomic<T>`'s default value.
  - It makes each call both a release operation and an acquire operation.

84

Compiler Intrinsics

- Here are versions of `my_atomic<T>`'s members that use intrinsic functions:

```
template <typename T>
T my_atomic<T>::store(T v) const {
    return __atomic_load_n(&value, __ATOMIC_SEQ_CST);
}

template <typename T>
void my_atomic<T>::store(T v) const {
    __atomic_store_n(&value, v, __ATOMIC_SEQ_CST);
}
```

- The compiler will automatically figure out whether to use a 1-, 2-, 4-, or 8-byte operation based on `value`'s data type.

85

Compiler Intrinsics

- GCC often provides additional intrinsic atomic functions such as:
  - `__atomic_exchange_n`
  - `__atomic_compare_exchange_n`
  - `__atomic_fetch_add`
- Other compilers often provide their own versions of these functions…

86

Compiler Intrinsics

- For example, MSVC provides these functions for the ARM64 platform:
  - `_InterlockedExchange`
  - `_InterlockedCompareExchange`
  - `_InterlockedAdd`
- By default, these are 32-bit operations.
- However, you can append an 8, 16, or 64 to the end to choose a different operand size, as in:
  - `_InterlockedExchange8`
  - `_InterlockedCompareExchange16`
  - `_InterlockedAdd64`

87

Takeaways

- Atomic operations must be non-preemptible.
- Atomic operations must synchronize with atomic operations on other threads.
  - There are fine-grained synchronization options that are beyond the scope of this session — see the `std::memory_order` enumeration for details. [cppreference.com (2025)]
- Compilers may provide intrinsic functions like GCC's `__atomic_*` functions to help you implement these operations.
- If not, you can implement these operations through embedded assembly blocks.

88

The End

Thanks for Listening

89

Bibliography

- Frasch, C. (2023, October 14). *CppCon/CPPCON2023: Slides and other materials from cppcon 2023*. GitHub. https://github.com/CppCon/CppCon2023
  - See Also: YouTube [Video]. https://youtu.be/K3P_Lmq6pw0
- *std::memory_order - cppreference.com*. (2025, Sept 13). cppreference.com. (n.d.). https://en.cppreference.com/w/cpp/atomic/memory_order.html

90