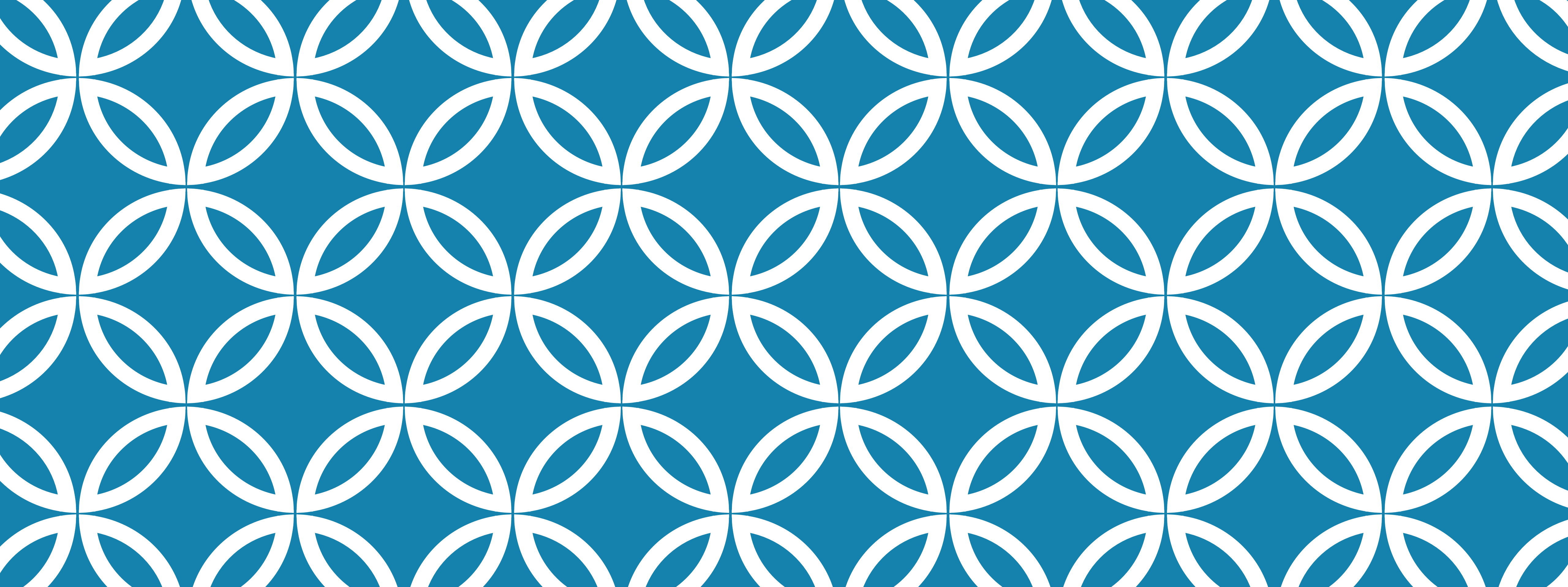


# MONITOR PATTERN

Professor Ken Birman

CS4414 Lecture 16

# IDEA MAP FOR TODAY

Reminder: Thread Concept

Lightweight vs. Heavyweight

Thread "context"

C++ mutex objects. Atomic data types.

The monitor pattern in C++

Problems monitors solve (and problems they don't solve)

Deadlocks and Livelocks

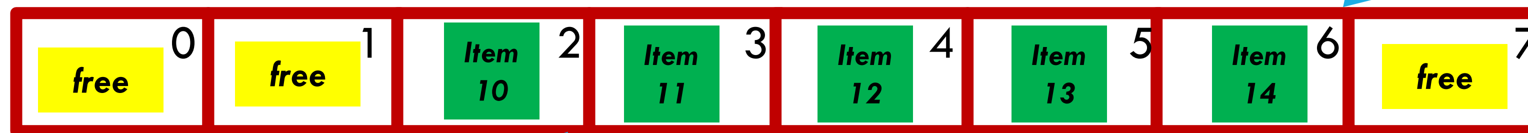
**Today we focus on monitors.**

# BOUNDED BUFFER: THE ABSTRACTION IS OF A RING. THE IMPLEMENTATION IS A FIXED SIZED ARRAY

We take an array of some fixed size, LEN, and think of it as a ring. The k'th item is at location (k % LEN). Here, LEN = 8

*Producers write to the end of the full section*

**nfree =3**
**free_ptr = 15**
*15 % 8 = 7*




*Consumers read from the head of the full section*

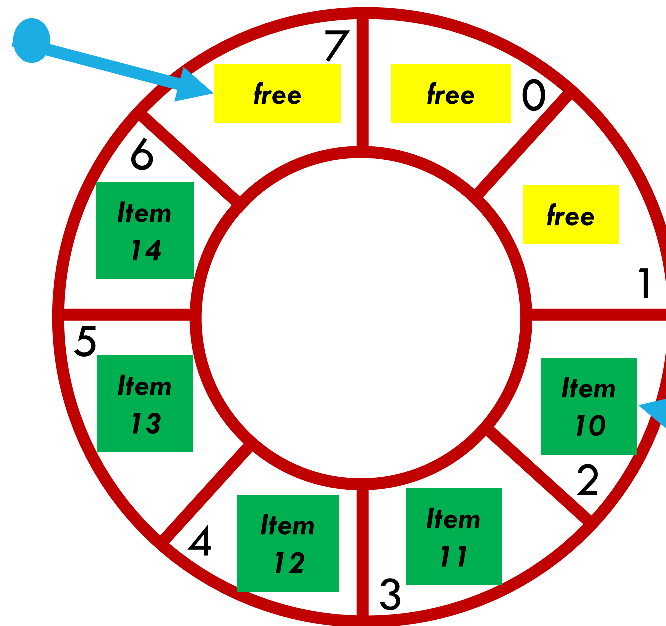
**nfull =5**
**next_item = 10**
*10 % 8 = 2*

# BOUNDED BUFFER: THE ABSTRACTION IS OF A RING. THE IMPLEMENTATION IS A FIXED SIZED ARRAY

Now, wrap this into a circle, with cell 0 next to cell 7. No other change is made – the remainder of the figure is identical.

*Producers write to the end of the full section*

**nfree =3**
**free_ptr = 15**
*15 % 8 = 7*




*Consumers read from the head of the full section*

**nfull =5**
**next_item = 10**
*10 % 8 = 2*

# A PRODUCER OR CONSUMER WAITS IF NEEDED

Producer:

```
void produce(const Foo& obj)
{

    if(nfull == LEN) wait;
    buffer[free_ptr++ % LEN] = obj;
    ++nfull;
    - - nempty;
}
```

Consumer:

```
Foo consume()
{

    if(nfull == 0) wait;
    ++nempty;
    - - nfull;
    return buffer[next_item++ % LEN];
}
```

As written, this code is unsafe… we can't fix it just by adding atomics or locks!

# A PRODUCER OR CONSUMER WAITS IF NEEDED

```
std::mutex mtx;
```

**Producer:**

```
void produce(const Foo& obj)
{
     std::scoped_lock plock(mtx);
     if(nfull == LEN) wait;
     buffer[free_ptr++ % LEN] = obj;
     ++nfull;
     - - nempty;
}
```

**Consumer:**

```
Foo consume()
{
     std::scoped_lock clock(mtx);
     if(nfull == 0) wait;
     ++nempty;
     - - nfull;
     return buffer[next_item++ % LEN];
}
```

Now safe… but lacks a way to implement "wait"

# WHY ISN'T IT TRIVIAL TO IMPLEMENT WAIT?

While holding one lock, a thread can't use *locking* to wait for some condition to hold: nobody could "signal" for it to wake up because no other thread can acquire the lock

But if we release the locks on the critical section, "anything" can happen! The condition leading to wanting to wait might vanish.

```
…
std::scoped_lock plock(mtx);
if(nfull == LEN) { release lock; wait; reacquire lock; }
…
```

Right here, before wait, context switch could occur

# WITH UNIQUE_LOCK, THERE IS A WAY TO DO A WAIT.

```
std::mutex mtx;
```

**Producer:**

```
void produce(const Foo& obj)
{
      std::unique_lock plock(mtx);
      if(nfull == LEN) wait;
      buffer[free_ptr++ % LEN] = obj;
      ++nfull;
      - - nempty;
}
```

**Consumer:**

```
Foo consume()
{
      std:: unique_lock clock(mtx);
      if(nfull == 0) wait;
      ++nempty;
      - - nfull;
      return buffer[next_item++ % LEN];
}
```

# THE MONITOR PATTERN

Our example turns out to be a great fit to the monitor pattern.

A monitor combines protection of a critical section with additional operations for waiting and for notification.

For each protected object, you will need a “mutex” object that will be the associated lock.

# A MONITOR IS A “PATTERN”

It uses a unique_lock to protect a critical section. You designate the mutex (and can even lock multiple mutexes atomically).

Monitor *conditions* are variables that a monitor can wait on:

- **wait** is used to wait. It also (atomically) releases the scoped_lock.
- **wait_until** and **wait_for** can also wait for a timed delay to elapse.
- **notify_one** wakes up a waiting thread… **notify_all** wakes up all waiting threads. If no thread is waiting, these are both no-ops.

# STD::SHARED_LOCK AND STD::UNIQUE_LOCK

We will discuss in a moment, but

- **std::shared_lock** is a form of read-lock. Multiple readers can acquire a shared_lock on the identical mutex.
- **std::unique_lock** is like **std::scoped_lock**: a form of write-lock. The difference is that std::scoped_lock is less costly but lacks a feature we need for monitors. std::unique_lock works for the monitor pattern. As for std::shared_lock, this is never used when implementing a monitor.

# SOLUTION TO THE BOUNDED BUFFER PROBLEM USING A MONITOR PATTERN

We will need a mutex, plus two “condition variables”:

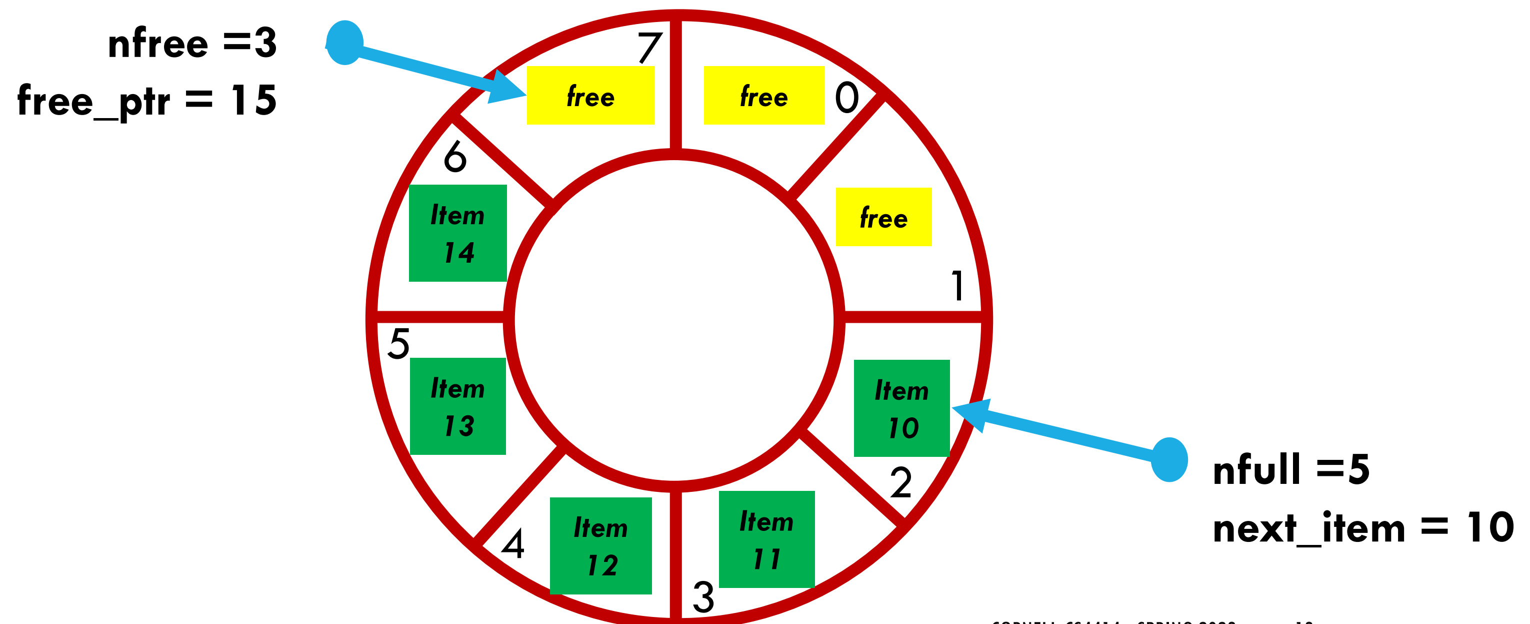
```
std::mutex mtx;
std::condition_variable not_empty;
std::condition_variable not_full;
```

… our code will have a single critical section with two roles (one to produce, one to consume), so we use one mutex.

# INITIALIZATION OF THE VARIABLES

First, we need our const int LEN, and int variables nfree, nfull, free_ptr and next_item. Initially everything is free: nfree = LEN;

```
const int LEN = 8;
int nfree = LEN;
int nfull = 0;
int free_ptr = 0;
int next_item = 0;
```



# INITIALIZATION OF TI

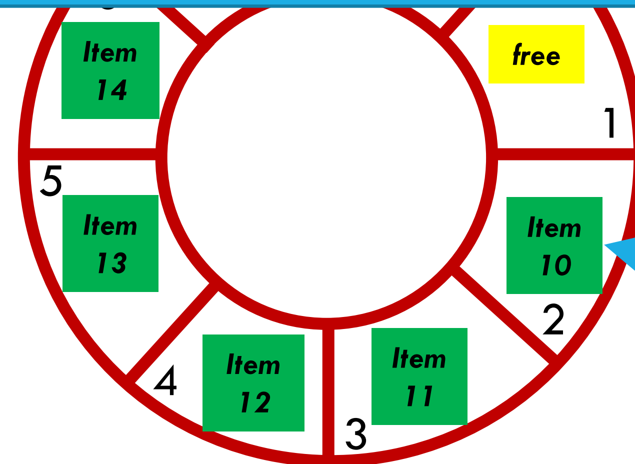
First, we need our const int LE
free_ptr and next_item. Initi

const int LEN = ;
int nfree = LEN;
int nfull = 0;
int free_ptr = 0;
int next_item = 0;

We don't declare these as atomic or volatile because we plan to only access them only inside our monitor!

Only use those annotations for "stand-alone" variables accessed concurrently without locking

free_ptr =




nfull =5
next_item = 10

# CODE TO PRODUCE AN ITEM

```
void produce(const Foo& obj)
{
        std::unique_lock plock(mtx);
        not_full.wait(plock, [&](){ return nfree != 0;});
        buffer[free_ptr++ % LEN] = obj;
        --nfree;
        ++nfull;
        not_empty.notify_one();
}
```

# CODE TO PRODUCE AN

```
void produce(const Foo&
{
        std::unique_lock plock(mtx);
        not_full.wait(plock, [&](){ return nfree != 0;});
        buffer[free_ptr++ % LEN] = obj;
        --nfree;
        ++nfull;
        not_empty.notify_one();
}
```

This lock is automatically held until the end of the method, then released. But it will be temporarily released for the condition-variable "wait" if needed, then automatically reacquired

# CODE TO PRODUCE AN ITEM

```
void produce(const Foo& obj)
{
        std::unique_lock plock(mtx);
        not_full.wait(plock, [&](){ return nfree != 0;});
        buffer[free_ptr++ % LEN] = obj;
        --nfree;
        ++nfull;
        not_empty.notify_one();
}
```

# CODE TO PRO

A condition variable implements wait in a way that atomically puts this thread to sleep and releases the lock. This guarantees that if notify should wake A up, A will “hear it”

When A does run, it will also automatically reaquire the mutex lock.

```
void produce(co
{
        std::uni    e_lo
        not_full.wait(plock, [&](){ return nfree != 0;});
        buffer[free_ptr++ % LEN] = obj;
        --nfree;
        ++nfull;
        not_empty.notify_one();
}
```

# CODE TO PRODUCE AN ITEM

```
void produce(const Foo& obj)
{
        std::unique_lock plock(mtx);
        not_full.wait(plock, [&](){ return nfree != 0;});
        buffer[free_ptr++ % LEN] = obj;
        --nfree;
        ++nfull;
        not_empty.notify_one();
}
```

# CODE TO PR

The condition takes the form of a lambda returning true or false. It checks "what you are waiting for", not "why you are waiting".

```
void produce(Foo obj)
{
        std::unique_lock plock(mtx);
        not_full.wait(plock, [&](){ return nfree != 0;});
        buffer[free_ptr++ % LEN] = obj;
        --nfree;
        ++nfull;
        not_empty.notify_one();
}
```

# CODE TO PRODUCE AN ITEM

```
void produce(const Foo& obj)
{
        std::unique_lock plock(mtx);
        not_full.wait(plock, [&](){ return nfree != 0;});
        buffer[free_ptr++ % LEN] = obj;
        --nfree;
        ++nfull;
        not_empty.notify_one();
}
```

# CODE TO PRODUCE AN ITEM

```
void produce(const Foo& obj)
{
        std::unique_lock plock(mtx);
        not_full.wait(plock, [&](){ return nfree != 0;});
        buffer[free_ptr++ % L
        --nfree;
        ++nfull;
        not_empty.notify_one();
}
```

We produced one item, so we only need to wake up one of the waiting threads

# CODE TO CONSUME AN ITEM

```
Foo consume()
{
        std::unique_lock clock(mtx);
        not_empty.wait(clock, [&]() { return nfull != 0; });
        ++nfree;
        --nfull;
        not_full.notify_one();
        return buffer[full_ptr++ % LEN];
}
```

# CODE TO CONSUME AN ITEM

```
Foo consume()
{
        std::unique_lock clock(mtx);
        not_empty.wait(cl
        ++nfree;
        --nfull;
        not_full.notify_one();
        return buffer[full_ptr++ % LEN];
}
```

The notify doesn't need to be the last line of the consume method – it still holds the mutex lock, so nobody else can enter the critical section

# CODE TO CONSUME AN ITEM

```
Foo consume()
{
        std::unique_lock clock(mtx);
        not_empty.wait(clock, [&]() { return nfull != 0; });
        ++nfree;
        --nfull;
        not_full.notify_one();
        return buffer[full_ptr++ % LEN];
}
```

For the same reason, this return statement is safe: C++ executes the expression used in this return statement while still holding the lock.

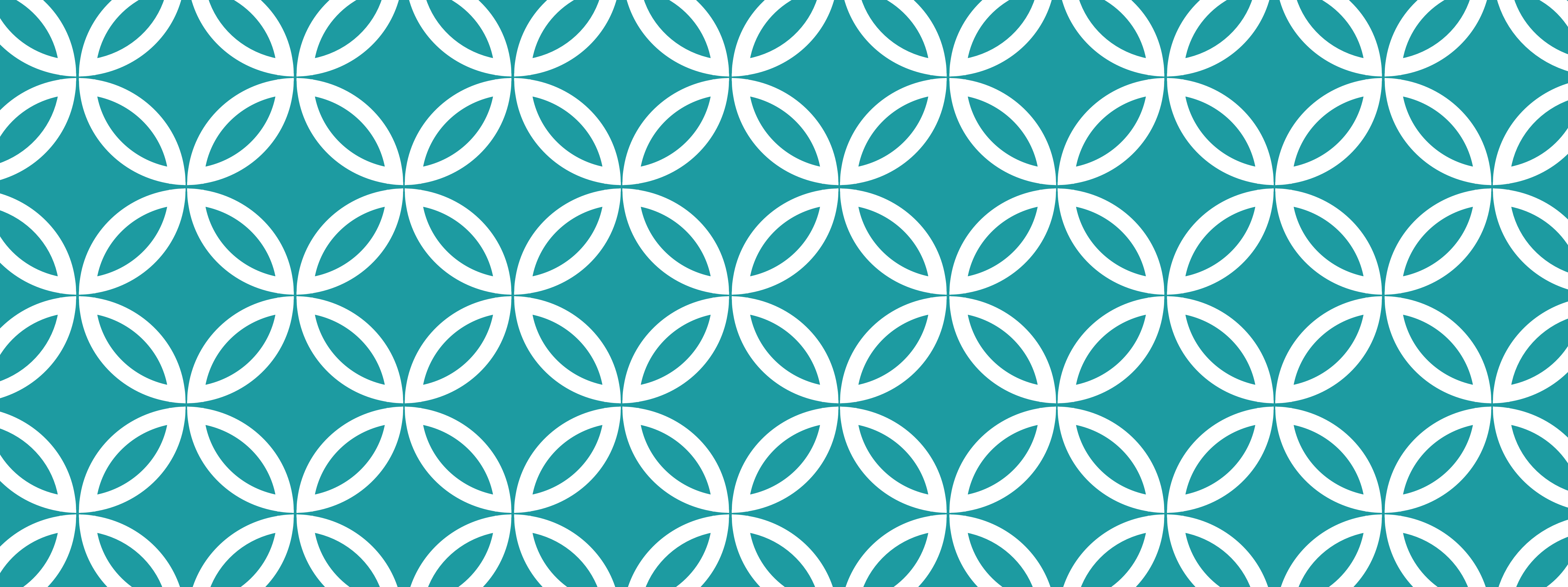
# CODE TO CONSUME AN ITEM

```
Foo consume()
{
        std::unique_lock clock(mtx);
        not_empty.wait(clock, [&]() { return nfull != 0; });
        ++nfree;
        --nfull;
        not_full.notify_one();
        return buffer[full_ptr++ % LEN];
}
```

# CODE TO CONSUME AN ITEM

```
Foo consume()
{
                                         = 0; });

      buffer[full_ptr++ % LEN];
}
```

This is where the scope is actually closed.  It happens as C++ performs the logic for actually returning the result (the Foo item "computed" by the return statement).  The destructor for <u>clock</u> now runs and releases the lock



# A SECOND EXAMPLE

## Readers and Writers

# RECALL THE RULE FOR SHARING A STD LIBRARY DATA STRUCTURE

A shared data structure can support arbitrary numbers of concurrent read-only accesses.

But an update (a “writer”) might cause the structure to change, so updates must occur when no reads are active.

We also need *fairness*: reads should not *starve* updates

# RECALL THE RULE FOR SHARING A STD LIBRARY DATA STRUCTURE

This can be solved using std::shared_lock and std::unique_lock: std::shared_lock is a read lock and std::unique_lock is a write lock.

But the default implementation allows readers to starve writers. A steady stream of readers would continuously acquire the shared reader lock. No writer could ever get in!

We can do better… with a monitor where we control the policy

# EXPRESSED AS A MONITOR

```
std::mutex mtx;
std::condition_variable want_rw;
int active_readers = 0, writers_waiting = 0;
bool active_writer = false;
```

```
void start_read()
{
      std::unique_lock srlock(mtx);
      want_rw.wait(srlock, [&]() { return ! ((active_writer || writers_waiting); });
      ++active_readers;
}

void end_read()
{
       std::unique_lock erlock(mtx);
       if(--active_readers == 0)
               want_rw.notify_all();
}
```

```
void start_write()
{
      std::unique_lock swlock(mtx);
      ++writers_waiting;
      want_rw.wait(swlock, [&]() { return !(active_writer || active_readers); });
      --writers_waiting;
      active_writer = true;
}

void end_write()
{
       std::unique_lock ewlock(mtx);
       active_writer = false;
       want_rw.notify_all();
}
```

# EXPRESSED AS A MONITOR

```
std::mutex mtx;
std::condition_variable want_rw;
int active_readers = 0, writers_waiting = 0;
bool active_writer = false;
```

```
void s
{
    st
    w
    ++active_readers;
}

void end_read()
{
    std::unique_lock erlock(mtx);
    if(- -active_readers == 0)
        want_rw.notify_all();
}
```

```
r || active_readers); });
}

void end_write()
{
    std::unique_lock ewlock(mtx);
    active_writer = false;
    want_rw.notify_all();
}
```

# EXPRESSED AS A MONITOR

std::mutex mtx;
std::condition_v
int active_reade
bool active_wri

C++ interprets this as **(writers_waiting > 0)**

```
void start_read()
{
    std::unique_lock srlock(mtx);
    want_rw.wait(srlock, [&]() { return ! ((active_writer || writers_waiting); });
    ++active_readers;
}

void end_read()
{
     std::unique_lock erlock(mtx);
     if(- -active_readers == 0)
            want_rw.notify_all();
}
```

vriter || active_readers); });

# EXPRESSED AS A MONITOR

std::mutex m
std::conditio
int active_reade
bool active_wri

“wait until there is no active writer and there are no waiting writers”

```
void start_read()
{
     std::unique_lock srlock(mtx);
     want_rw.wait(srlock, [&]() { return ! ((active_writer || writers_waiting); });
     ++active_readers;
}

void end_read()
{
      std::unique_lock erlock(mtx);
      if(- -active_readers == 0)
            want_rw.notify_all();
}
```

vriter || active_readers); });

# … USING LAMBDAS

```
std::mutex mtx;
std::condition_variable want_rw;
int active_readers, writers_waiting;
bool active_writer;
```

```
void start_read()
{
      std::unique_lock srlock(mtx);
      want_rw.wait(srlock, [&]() { return !
      ++active_readers;
}

void end_read()
{
       std::unique_lock erlock(mtx);
       if(- -active_readers == 0)
              want_rw.notify_all();
}
```

```
void st
{
      std::unique_lock swlock(mtx);
      + +writers_waiting;
      want_rw.wait(swlock, [&]() { return !(active_writer | | active_readers); });
      - -writers_waiting;
      active_writer = true;
}

void end_write()
{
       std::unique_lock ewlock(mtx);
       active_writer = false;
       want_rw.notify_all();
}
```

C++ interprets this as **(active_readers > 0)**

# … USING LAMBDAS

```
std::mutex mtx;
std::condition_variable
int active_readers, write
bool active_writer;

void start_read()
{
      std::unique_lock srlock(mtx);
      want_rw.wait(srlock, [&]() { return
      ++active_readers;
}

void end_read()
{
       std::unique_lock erlock(mtx);
       if(- -active_readers == 0)
              want_rw.notify_all();
}
```

"wait until there is no active writer and there are no active readers"

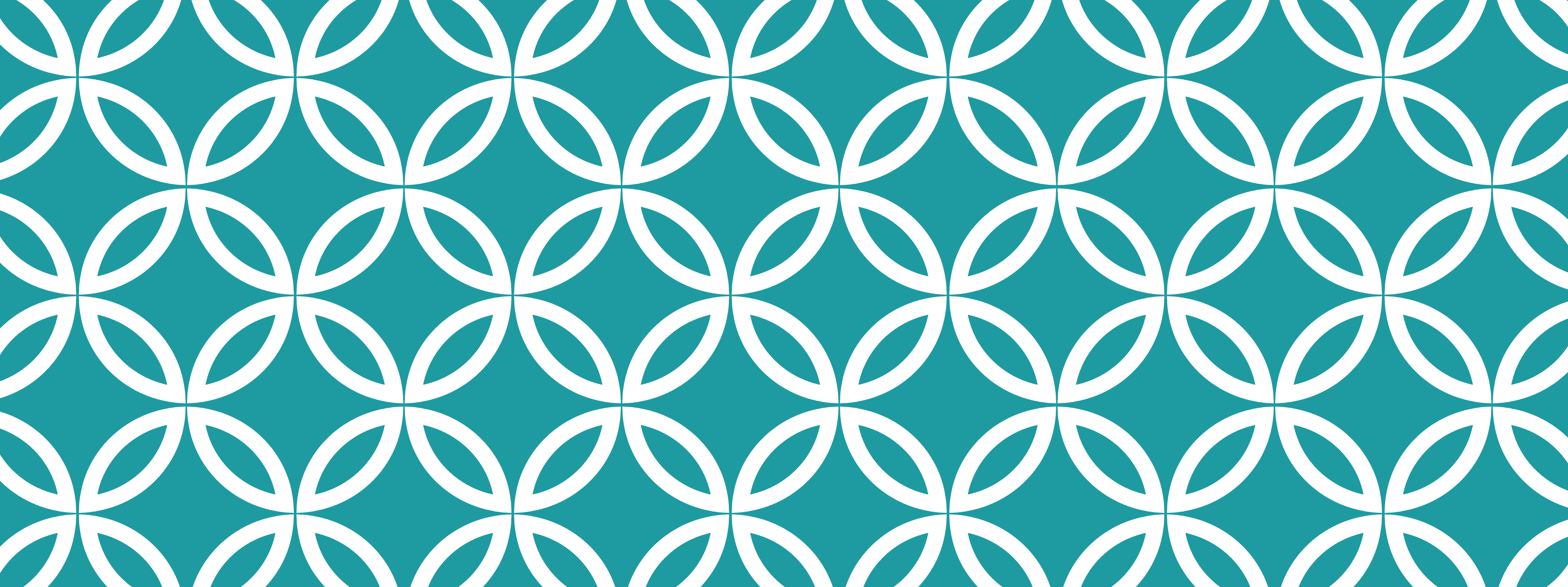
```
{
      std::unique_lock s      k(mtx);
      + +writers_waiting;
      want_rw.wait(swlock, [&]() { return !(active_writer | | active_readers); });
      - -writers_waiting;
      active_writer = true;
}

void end_write()
{
       std::unique_lock ewlock(mtx);
       active_writer = false;
       want_rw.notify_all();
}
```

# COOL IDEA – YOU COULD EVEN OFFER IT AS A PATTERN…

```
beAReader([](){ … some code to execute as a reader });
```

```
beAWriter([](){ … some code to execute as a writer });
```

All beAReader would do is call start_read, then call the lambda, then end_read. Same for beAWriter: call start_write, then the lambda, then end_write.



# A FEW THINGS TO NOTE

**Monitor features that matter when coding with them**

# OUR ULTIMATE VERSION OF READERS AND WRITERS IS SIMPLE AND CORRECT.

But it gives waiting writers priority over waiting readers, so it isn't fair (an endless stream of writers would starve readers).

In effect, we are assuming that writing is less common than reading. You can modify it to have the other bias easily (if writers are common but readers are rare).

# OUR ULTIMATE VERSION OF READERS AND WRITERS IS SIMPLE AND CORRECT.

Readers yield to writers, even if they are waiting

```
void start_read()
{
    std::unique_lock srlock(mtx);
    want_rw.wait(srlock, [&]() { return ! ((active_writer || writers_waiting); });
    ++active_readers;
}

void end_read()
{
    std::unique_lock erlock(mtx);
    if(- -active_readers == 0)
        want_rw.notify_all();
}
```

B... rs, so it is... aders).

In... han re... ly (if w...

# OUR ULTIMATE VERSION OF READERS AND WRITERS

But it gives w
isn't fair (an

In effect, we
reading. You
writers are c

Writers don't yield to *waiting readers*

```
void start_write()
{
    std::unique_lock swlock(mtx);
    + +writers_waiting;
    want_rw.wait(swlock, [&]() { return !(active_writer | | active_readers); });
    - -writers_waiting;
    active_writer = true;
}

void end_write()
{
    std::unique_lock ewlock(mtx);
    active_writer = false;
    want_rw.notify_all();
}
```

# NOTIFY_ALL VERSUS NOTIFY_ONE

notify_all wakes up *every* waiting thread. We used it here, because sometimes the next thread to enter should be a reader and sometimes a writer.

One can be fancy and use notify_one to try and make this code more fair, but it isn't easy to do because your solution would still need to be correct with spurious wakeups.

# OUR ULTIMATE VERSION OF READERS AND WRITERS IS SIMPLE AND CORRECT.

What we just saw:

- The readers wait even if there is a waiting writer. So if there is an active writer or a waiting writer, a reader pauses in start read.
- A writer only waits if there is an active writer or an active reader. If a writer wants to start writing and nobody is active it gets in *before any reader would be able to start reading*.
- *This is what we mean by "prioritizes writers over readers".*

# IS PRIORITIZING WRITERS A GOOD IDEA?

If you expect a high rate of readers and a low rate of writers it makes sense.

Presumably you want your application to always see updates as soon as possible.

But if you have a very high rate of writes, readers starve.

# IN FACT, A SYMMETRIC VERSION IS FEASIBLE!

It is a bit more complicated and doesn't fit on one slide

The basic idea is this: new readers will prioritize "switching" to a writer, if one is waiting. But if an active writer calls end_write and there is a reader waiting, let all the readers in before the next writer can enter.

We won't show it (but copilot would show you the code if asked).

# WARNING ABOUT "SPURIOUS WAKEUPS"

We do not recommend using the condition-variable **wait** method without a lambda. It supports this, but your code would *need to use a while loop* and retest the wait condition if you do that.

The reason? *Wait can sometimes wake up even when notify was not called.* This is a documented feature but means you must always recheck the condition. The lambda version of **wait** does so.

# DEBUGGING SOMEONE ELSE'S BUGGY MONITOR CODE? CHECK FOR THIS FIRST!

Always check their condition-wait logic. Recall our start_write:

```
void start_write()
{
     std::unique_lock swlock(mtx);
     + +writers_waiting;
     want_rw.wait(swlock, [&]() { return !(active_writer | | active_readers); });
     - -writers_waiting;
     active_writer = true;
}
```

# DEBUGGING SOMEONE ELSE'S BUGGY MONITOR CODE? CHECK FOR THIS FIRST!

Old-fashioned version from CS4410 has a while loop:

```
void start_write()
{
    std::unique_lock swlock(mtx);
    + +writers_waiting;
    while(active_writer | | active_readers)
        want_write.wait(swlock | | active_readers); });
    - -writers_waiting;
    active_writer = true;
}
```

# DEBUGGING SOMEONE ELSE'S BUGGY MONITOR CODE? CHECK FOR THIS FIRST!

Old-fashioned version fr

In the CS4410 version, there were two condition variables, one for a waiting writer, one for a reader

```
void start_write()
{
      std::unique_lock swlo
      + +writers_waiti
      while(active_ riter | | active_readers)
            want_write.wait(swlock | | active_readers); });
      - -writers_waiting;
      active_writer = true;
}
```

# DEBUGGING SOMEONE ELSE'S BUGGY MONITOR CODE? CHECK FOR THIS FIRST!

In fact, if you really look at your old CS4410 notes you actually might find this:

```
void start_write()
{
      std::unique_lock swlock(mtx);
      + +writers_waiting;
      if(active_writer | | active_readers)
          want_write.wait(swlock | | active_readers); });
      - -writers_waiting;
      active_writer = true;
}
```

# DEBUGGING SOMEONE ELSE'S BUGGY MONITOR CODE? CHECK FOR THIS FIRST!

In fact, if you really look
might find this:

```
void start_write()
{
      std::unique_lock        ck(mtx);
      + +writ      _waiting;
      if(active_writer | | active_readers)
           want_write.wait(swlock | | active_readers); });
      - -writers_waiting;
      active_writer = true;
}
```

This **if** statement is evaluated once. Then perhaps we wait.

# WOULD THIS BUG MATTER? YOU BET!

Our lambda version didn't have a bug. Neither did the version with the while-loop. The **if** version is ok, in C

But the **if** is a (surprisingly common) mistake in C++. People don't realize that wait might wake up even without a notify_one.

**Consequence? A new writer starts when other threads are still in the critical section, violating the std library policy!**

# DEBUGGING SOMEONE ELSE'S BUGGY MONITOR CODE? CHECK FOR THIS FIRST!

In fact, if you really look at your old CS4410 notes you actually might find this:

**This wait might wake up for the <u>wrong reason</u>, not because notify_one was called**

```
void start_write()
{
      std::unique_lock swlock(mtx);
      + +writers_waiting;
      if(active_writer | | active_readers)
            want_write.wait(swlock | | active_readers); });
      - -writers_waiting;
      active_writer = true;
}
```

# FAIRNESS, FREEDOM FROM STARVATION

Locking solutions for NUMA system map to atomic “test and set”:

```
std::atomic_flag lock_something = ATOMIC_FLAG_INIT;

  while (lock_something.test_and_set()) {}      // Threads loop waiting, here

  cout << “My thread is inside the critical section!” << endl;

  lock_stream.clear();
```

This is random, hence “fair”, but not *guaranteed* to be fair.

# HOW COULD TEST_AND_SET BE UNFAIR?

On a NUMA machine, the mutex has to be near some core or in DRAM. Suppose it gets allocated in the memory close to core 0.

Now suppose thread A is on core 0 competing with threads B and C on cores 1 and 2. Due to NUMA effects (lecture 2), A can access the mutex 5x faster than threads B and C!

So thread A will have an unfair advantage

# BASICALLY, WE COULD WORRY ABOUT FAIRNESS, BUT DIDN'T IN THIS EXAMPLE

Our home-brew "lock implementation" was thus unfair.

The std::unique_lock implementation used in monitors tries to be much more fair, but NUMA effects could still "defeat" it!

This is just something to be aware of. Ideally you would want all your threads close to the mutex, or none of them close to it.

# KEEP LOCK BLOCKS SHORT

It can be tempting to just get a lock and then do a whole lot of work while holding it.

But keep in mind that if you really needed the lock, some thread may be waiting this whole time!

So… you'll want to hold locks for as short a period as feasible.

# RESIST THE TEMPTATION TO RELEASE A LOCK WHILE YOU STILL NEED IT!

Suppose threads A and B share:

```
std::map<std::string, int> myMap;
```

Now, A executes:

```
auto item = myMap[some_city];
cout << " City of " << item.first << ", population = " << item.second << endl;
```

Are both lines part of the critical section?

# HOW TO FIX THIS?

We can protect both lines with a scoped_lock:

```
std::mutex mtx;
….
{
        std::scoped_lock lock(mtx);
        auto item = myMap[some_city];
        cout << " City of " << item.first << ", population = " << item.second << endl;
}
```

# … BUT THIS COULD BE SLOW

Holding a lock for long enough to format and print data will take a long time.

Meanwhile, no thread can obtain this same lock.

# TYPICAL WORK-AROUND PEOPLE EXPLORE: PRINT OUTSIDE THE SCOPE

Tempting change:

```
std::mutex mtx;
std::pair<std::string,int> item;
{
        std::scoped_lock lock(mtx);
        item = myMap[some_city];
}
cout << " City of " << item.first << ", population = " << item.second << endl;
```

… this a correct piece of code. But this item could change even before it is printed.

# ONE IDEA: PRINT OUTSIDE THE SCOPE

Tempting change:

```
std::mutex mtx;
std::pair<std::string,int> *item;
{
        std::scoped_lock lock(mtx);
        item = &myMap[some_city];
}
cout << " City of " << item→first << ", population = " << item → second << endl;
```

Item might have been deleted by the time we try to print it. Our pointer could point to outer space!

This version is wrong! Can you see the error?

# BUT NOW THE PRINT STATEMENT HAS NO LOCK

No! This change is unsafe, for two reasons:

- Some thread could do something replace the std::pair that contains Ithaca with a different object. A would have a "stale" reference.
- Both std::map and std::pair are implemented in a non-thread-safe libraries. *If any thread could do any updates, a reader must view the whole structure as a critical section!*

# HOW DID FAST-WC HANDLE THIS?

In fast-wc, we implemented the code to never have concurrent threads accessing the same std::map!

Any given map was only read or updated by a single thread.

This does assume that std::map has no globals that somehow could be damaged by concurrent access to different maps, but in fact the library does have that guarantee.

# ARE THERE OTHER WAYS TO HANDLE AN ISSUE LIKE THIS?

A could safely make a copy of the item it wants to print, exit the lock scope, then print from the copy. It could even generate a vector of items to print "later", which is a common way to log debug data.

We could use two levels of locking, one for the map itself, a second for std::pair objects in the map.

We could add a way to "mark" an object as "in use by someone" and write code to not modify such an object.

# BUT BE CAREFUL!

The more subtle your synchronization logic becomes, the harder the code will be to maintain or even understand.

Simple, clear synchronization patterns have a benefit: anyone can easily see what you are doing!

This often causes some tradeoffs between speed and clarity.

# SYNCHRONIZATION SUMMARY

**atomic<t>** for base types (bool, int, float), test-and-set… No need to say "volatile" because the compiler infers that.

**scoped_lock** for most locking. Can lock multiple mutexes atomically.

**monitor pattern**: combines a **unique_lock** with condition variables to offer protection as well as a wait and notify mechanism, easy to reason about even for complex logic like reads/writers.