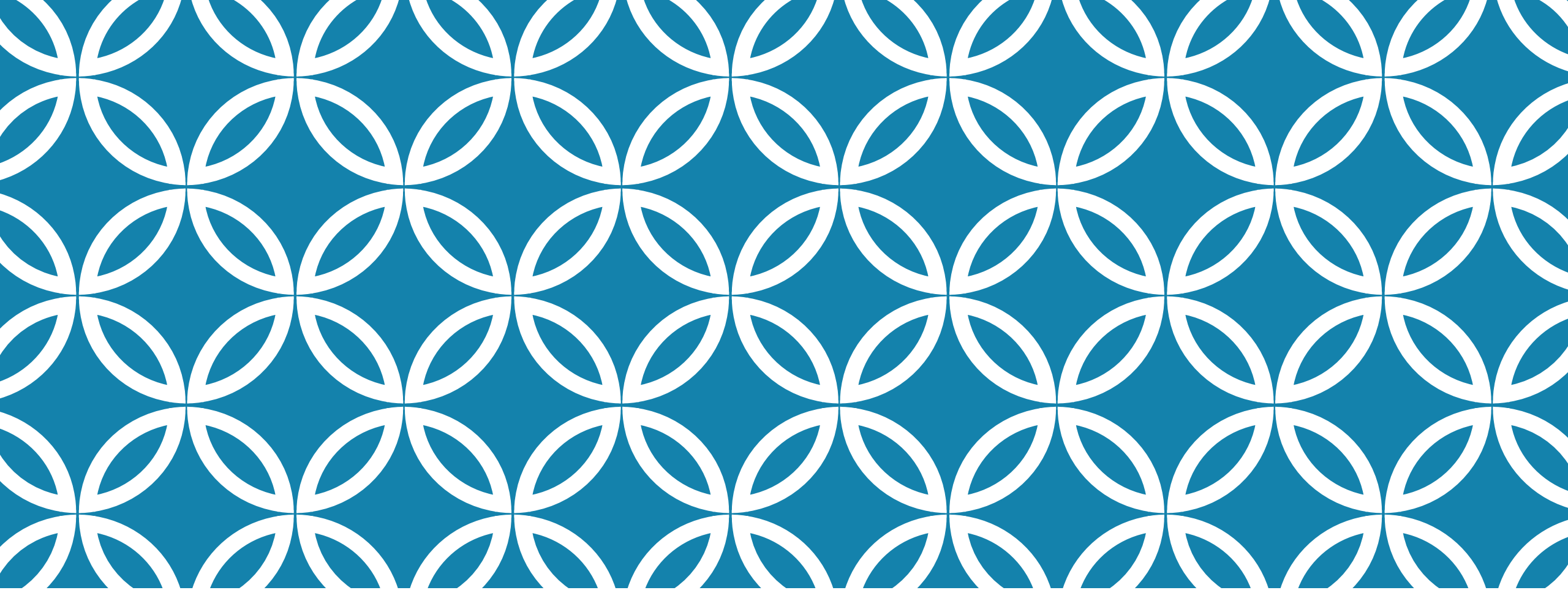


MONITOR PATTERN

Professor Ken Birman

CS4414/5416 Lecture 9

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)

C++ supports every imaginable kind of synchronization pattern. Monitors are our pick for complex settings.

C++ ATOMICS

If a single variable needs to be accessed atomically when shared by threads, but without locks, we use std::atomic<T>. The type T must be a native data type.

For example, std::atomic<int> is a safe counter that requires no additional locking. But it only protects operations against the variable, not expressions.

Example: even if X and Y are both std::atomic, X < Y isn't the same as doing comparison in a critical section (while holding a mutex).

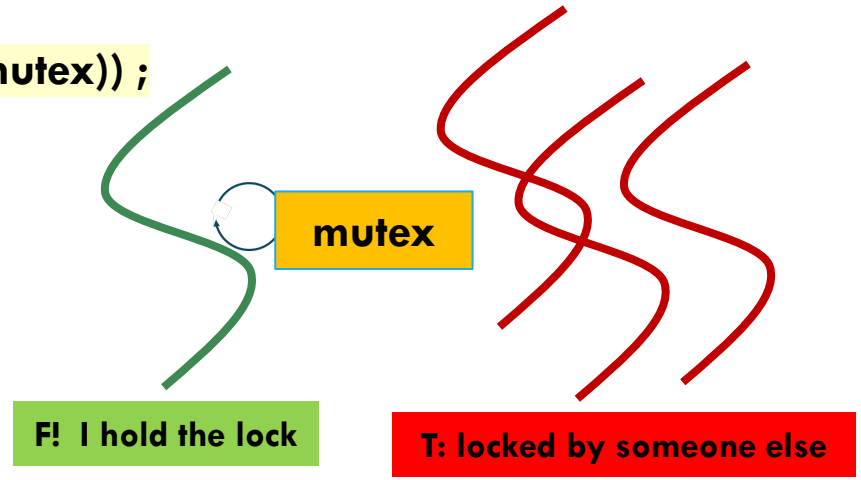
C++ ATOMICS

If you use std::atomic, you won't need volatile

std::atomic forces the *hardware and compiler to reread the variable on each access*. Prevents use of cached or speculated values.

UNDER THE HOOD

```
while(test-and-set(mutex)) ;
```



Mutex locking is done using a std::atomic<boolean>, initially **false (F).**

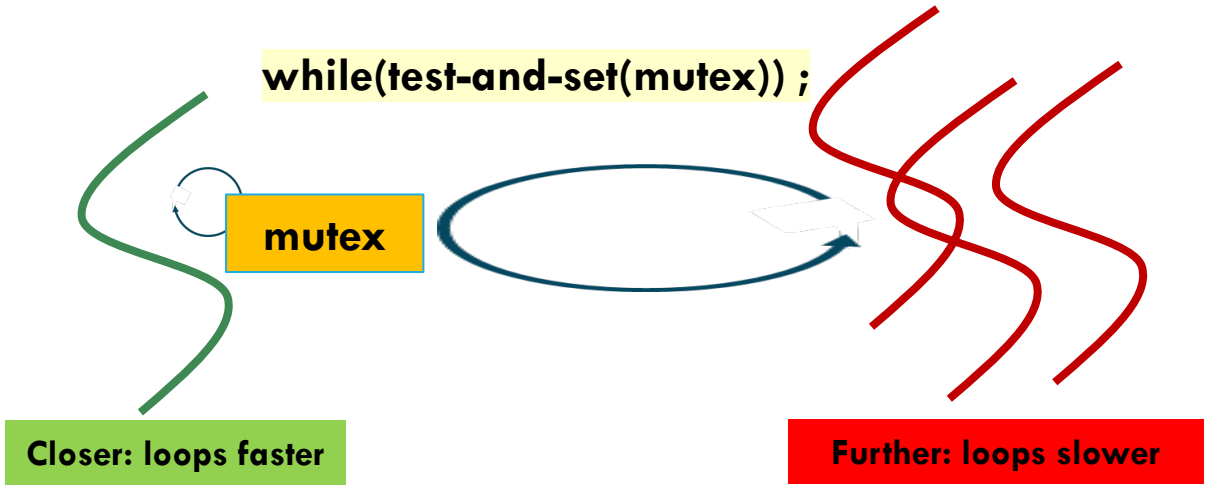
Threads desiring a lock loop perform an atomic operation defined this way:

```
test-and-set(x) { temp = x; x = T; return temp; }    // Compiles to a special instruction
```

- Returns **F** ⇒ it "won the race" and now holds the lock.
- Returns **T** ⇒ some other thread currently holds the lock.

To release a lock, the holder simply sets the mutex back to **F**.

UNDER THE HOOD



Competing to lock a mutex is not necessarily fair!

- Recall that memory is NUMA! Suppose the mutex is right next to thread A and far from threads B, C and D.
- Because thread A is close to the variable, it can access the memory location much faster. It loops more rapidly, and has an advantage
- In practice, few programs run into issues. C++ also has fair locking, but it is much slower and rarely used.

HOW TO SAFELY LOCK, THEN UNLOCK A MUTEX

Best is to do so in a block of code using std::shared_lock

Notice that this code gave the lock a name (cslock) and yet never uses the cslock variable

The reason is tied to the rule for when deconstructors run!

```
std::mutex mtx;

void critical_section()
{
    std::scoped_lock cslock(mtx);
    … do stuff, I hold the lock …
}
```

HOW TO SAFELY LOCK, THEN UNLOCK A MUTEX

Best is to do so in a block of code using std::shared_lock

Notice that this code gave the lock a name (cslock) and yet never uses the cslock variable

The reason is tied to the rule for when deconstructors run!

```
std::mutex mtx;

void critical_section()
{
    std::scoped_lock cslock(mtx);
    … do stuff, I hold the lock …
}
```

cslock is in scope until this block exits, so the lock is held until here

WHAT HAPPENS WITH CAUGHT EXCEPTIONS?

Suppose that while holding a lock, some method you call throws an exception and it is caught in a scope above where you acquired the lock?

With scoped_lock your lock will release. With a "hand acquired" lock you'll still hold it, but might easily forget that you do and deadlock… against yourself!

…. THOUGHT PUZZLE

What would have gone wrong with this version?

```
std::mutex mtx;

void critical_section()
{
      std::scoped_lock (mtx);
      … do stuff, I hold the lock …
}
```

…. THOUGHT PUZZLE

What would have gone wrong with this version?

A very common error!

C++ won't report the mistake but might warn with -Wall

```
std::mutex mtx;

void critical_section()
{
    std::scoped_lock (mtx);
    … do stuff, I don't hold the lock …
}
```

Lacking a variable name, acquires but *instantly releases* the lock

STD::SHARED_LOCK AND STD::UNIQUE_LOCK

std::scoped_lock implements true critical sections and is fast

In the examples we will look at next, we need more control

- **std::shared_lock** is a form of read-lock. Multiple readers can acquire a std::shared_lock on the identical mutex.
- **std::unique_lock** is the counterpart of std::shared_lock: a write-lock.
- Both also support a fancy form of waiting

STD::ATOMIC FOR CLASS TYPES

One experimental proposal for C++ extends std::atomic<T> but is limited to “trivially copyable” C++ classes!

Concurrent accesses are automatically handled safely: each operation looks like a transaction!

The technique uses no locks (but does a lot of copying)

How to use std::atomic<> effectively for non-primitive types

EXAMPLE PATTERN ONE: READERS/WRITERS

Very important stuff

READERS AND WRITERS

The default in C++ is that a std:: 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 a limited kind of *fairness*: an endless stream of reads should not *starve* (block) occasional updates

READERS AND WRITERS PATTERN

```
std::mutex mtx;

... block of code to do a read action
{
     std::shared_lock srlock(mtx);
     ... do your reading here ...
}

... block of code to do a write action
{
     std::unique_lock wrlock(mtx);
     ... do your writing here
}
```

READERS AND WRITERS AS METHODS

```
be_a_reader([&](){ … logic used by the reader });

be_a_writer([&](){ … logic used by the writer });
```

```
void be_a_reader(const std::function<void()>& read_action)
{
    std::shared_lock srlock(mtx);
    read_action();
}

void be_a_writer(const std::function<void()>& write_action)
{
     std::unique_lock wrlock(mtx);
     write_action();
}
```

HOW TO USE THE SECOND VERSION…

```
be_a_reader([&](){ … logic used by the reader });
```

```
be_a_writer([&](){ … logic used by the writer });
```

Uses variables from the caller scope by reference

When invoked, expects no arguments

Code, like for any method

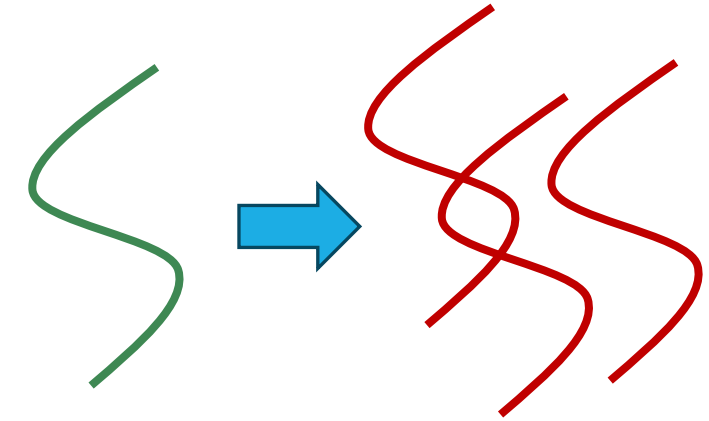
EXAMPLE PATTERN TWO: CIRCULAR BUFFER

Even more important stuff

CIRCULAR BUFFERS: A TOOL FOR THREAD-THREAD COMMUNICATION

When we created word count, we commented that the fastest version involved a main thread that launched word-counter threads, and a separate file opener thread.

How should the file opener thread "talk" to the word counter threads?



ASPECTS TO CONSIDER

In Linux, each open file has an associated *file descriptor*. This is a small integer. At most **ulimit()** files can be open at once.

C++ file reading normally uses a buffered I/O stream library such as std::ifstream. The buffer takes up space, which is another reason we limit how many files can be open at once.

Meanwhile inside the O/S, opening a file involves finding the name in a folder, getting the corresponding inode number, fetching the inode.

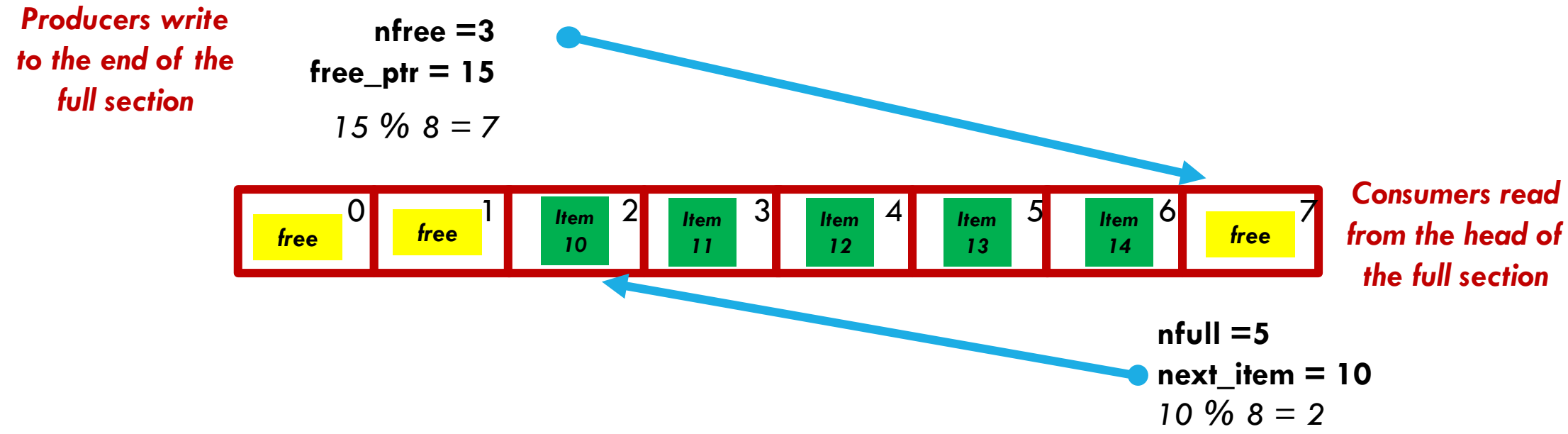
A GOOD SOLUTION

We wish to do this work "in anticipation of need"

- The separate file opening thread will pre-open a few files. It can be smaller than **ulimit()**, because each word counter will only scan one file at a time.
- We can already package the file descriptor into an ifstream, so the word counter won't have to do that.
- The image to have is of a "bucket brigade" for a fire. One thread fills buckets. Other threads dump water on the fire.

CIRCULAR BUFFER IMPLEMENTS THIS PATTERN

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

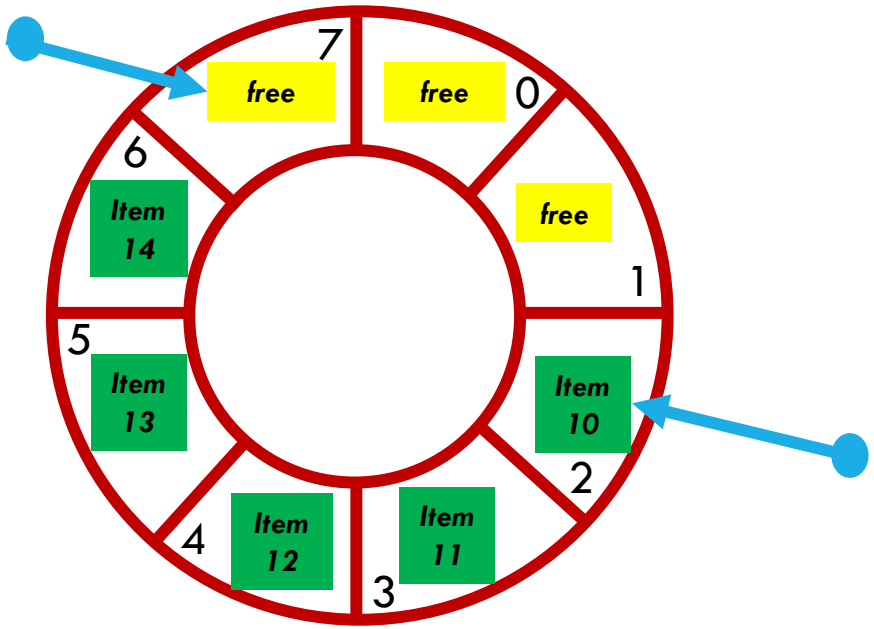


CIRCULAR BUFFER IMPLEMENTS THIS PATTERN

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;
}
```

Consumer:

```
Foo consume()
{


    if(nfull == 0) wait;
    - - 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::unique_lock plock(mtx);
    if(nfull == LEN) wait;
    buffer[free_ptr++ % LEN] = obj;
    ++nfull;
}
```

Consumer:

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

Now safe… but we still need to implement "wait"

WHY DID WE SWITCH FROM SCOPED_LOCK TO UNIQUE_LOCK (WHICH IS SLOWER)?

We can't just wait while holding the lock – nobody else can enter the critical section to consume something from the buffer.

But if we *release* the lock some other thread can instantly grab it

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

Right here, before wait, context switch could occur

Now *the buffer wouldn't be full anymore, yet the producer waits… forever.* Also causes a deadlock!

WITH UNIQUE_LOCK, THERE IS A SAFE WAY TO 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;
}
```

Consumer:

```
Foo consume()
{
      std:: unique_lock clock(mtx);
      if(nfull == 0) wait;
      - - 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 scoped_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.

SOLUTION TO THE BOUNDED BUFFER PROBLEM USING A MONITOR PATTERN

We will need a mutex, plus two "condition variables":

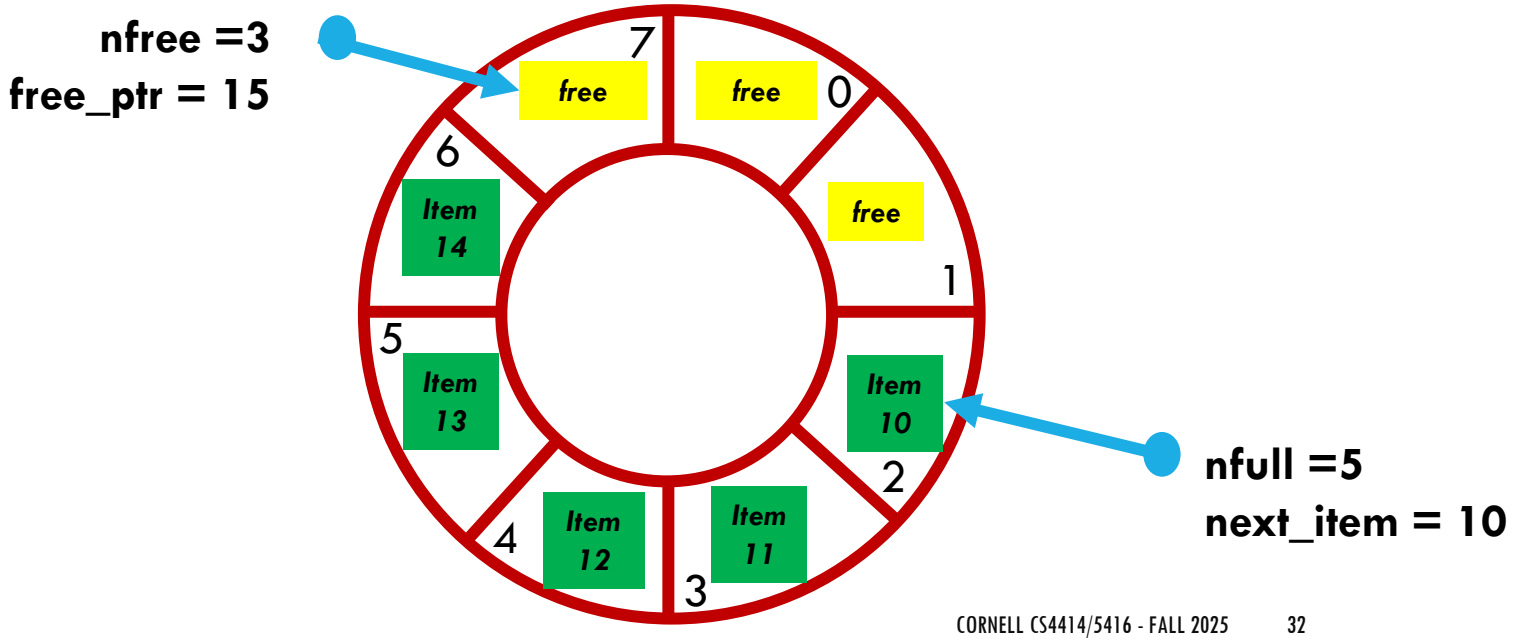
```
std::mutex mtx;
std::condition_variable not_empty;
std::condition_variable not_full;
```

… 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;
```



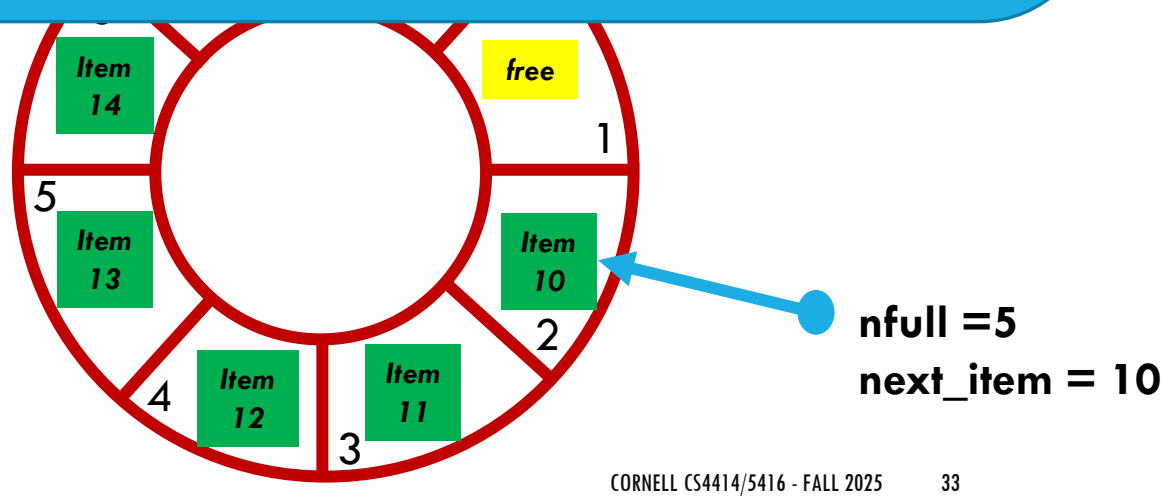
INITIALIZATION OF TH

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



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

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

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.

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()
{
    std::unique_lock<std::mutex> clock(mtx);
    while (full_ptr == empty_ptr) { cv.wait(clock); }
    Foo item = buffer[full_ptr++ % LEN];
    cv.notify_one();
    return item;
}
```

= 0; });

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 clock now runs and releases the lock

TEMPLATED AND WITH A STD::DEQUE

Why not create a templated class so that one implementation covers all uses?

- Then use a std::deque for the circular buffer and only check the size() to see if it is empty.
- … but one caveat: a fast producer could create unlimited data. So, do keep the capacity limit.

TEMPLATED AND WITH A STD::DEQUE

```
#include <mutex>
#include <deque>
#include <condition_variable>
```

```
template<typename T, const int LEN>
class Monitor {
    std::mutex mtx;
    std::condition_variable not_full, not_empty;
    std::deque buffer;

public:
    void produce(const T& obj) {
        std::unique_lock plock(mtx);
        not_full.wait(plock, & { return buffer.size() < LEN; });
        buffer.push_back(obj);
        not_empty.notify_one();
    }
```

```
    T consume() {
        std::unique_lock clock(mtx);
        not_empty.wait(clock, & { return !buffer.empty(); });
        T ret = buffer.front();
        buffer.pop_front();
        not_full.notify_one();
        return ret;
    }
};
```

RANDOM EXTRA STUFF

Such stuff!

A FEW REMARKS

notify_one wakes up one thread, notify_all wakes all of them up.

… but, due to "spurious wakeups" you cannot assume that each wakeup is tied to a specific notify_one.

Monitors do not *guarantee* fairness or even freedom from starvation. But in practice nobody ever runs into problems.

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.

CONSIDER THIS CODE… SUPPOSE IT NEEDS A LOCK

How would we know it needs one?

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

We need a lock if myMap or even this particular item could be modified. This code is reading the objects and if there are also writers, locking is needed.

WHAT ABOUT THIS VERSION?

Consider this protected critical section:

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

The code is correct and safe, but doing a print while holding the lock is going to be very slow

WHAT ABOUT THIS VERSION?

Better :

```
std::mutex mtx;
std::list<City> ilist;
        …. in the inner loop …
        {
                std::scoped_lock lock(mtx);
                auto item = myMap[some_city];
                ilist.push_back(item);                  // Makes a copy “freezing” the state we saw at this instant
        }
        …. later …
        for(auto item: ilist)
                cout << “ City of “ << item.first << “, population = “ << item.second << endl;
```

HOW DO PEOPLE WORK AROUND THIS?

The idea here is to create a kind of log to print later, updating it **while still inside the critical section.**

This way at the moment you did the list append, the data was definitely right there, and you took a "snapshot" by making a copy while you still held the lock.

Later you print the log outside the critical section.

BUT BE CAREFUL!

Yikes!
Avoid bad stuff!

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.

QUESTIONS FOR SELF-TEST

Suppose the producer is much faster than the consumers. What happens?

Suppose the buffer empties out. Now what happens?

Would be ideal to have a huge buffer with a tremendous number of produced items in it?

QUESTIONS FOR SELF-TEST

How do NUMA memory speeds create unfairness when using a shared atomic mutex?

If you know which memory unit the mutex is in, does this tell you which threads will get unfairly quick access, and which will be unfairly slow?

What would be a way to take control and eliminate this NUMA effect *without modifying the monitor pattern itself*?

QUESTIONS FOR SELF-TEST

Our readers and writers solution is asymmetric (new readers always wait to allow writers to run first).

Why do you think this has been popular and seen as a good choice (as distinct from using a symmetric monitor)?

What assumptions does it reflect about the computations readers and writers are doing?

QUESTIONS FOR SELF-TEST

In theory, with an endless stream of readers, std::shared_lock could starve std::unique_lock: writers would never get in.

The issue is considered implementation-dependent and unlikely

As a self-test, implement a monitor that can't have this issue.

QUESTIONS FOR SELF-TEST

We wanted an elegant, simple, high performance solution to problems like word count. List as many features of the program as you can that contribute to these goals.

Would you have thought of designing the code this way at the start? How can you develop mental patterns that lead directly to great solutions? Hint: understand the ideas, but practice using them!