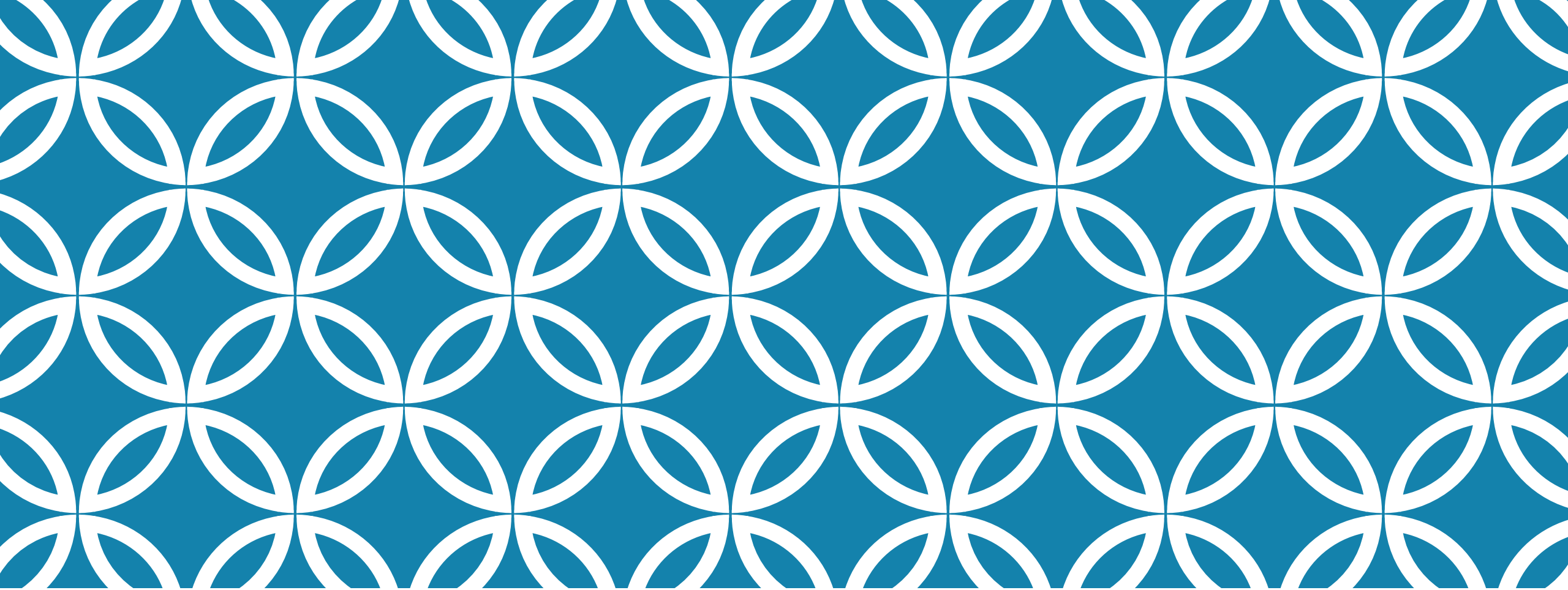


MONITOR PATTERN

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)

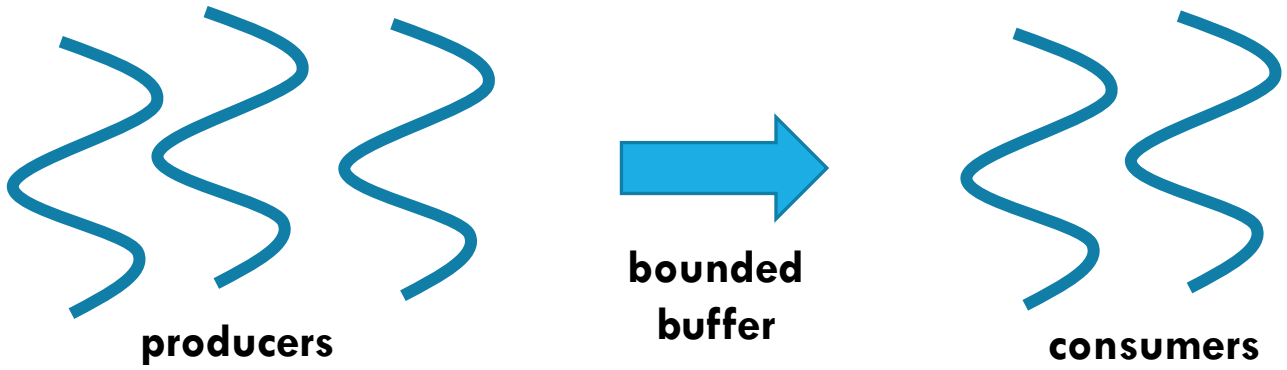
Deadlocks and Livelocks

Today we focus on monitors.

RAN OUT OF TIME TALKING ABOUT THREADS

Sometimes we can't pass information to child threads when we launch them, and are forced to pass information at runtime.

Bounded buffers are one of the best ways to do this.



REMINDER: A BOUNDED BUFFER

This example illustrates a famous pattern in threaded programs: the *producer-consumer* scenario

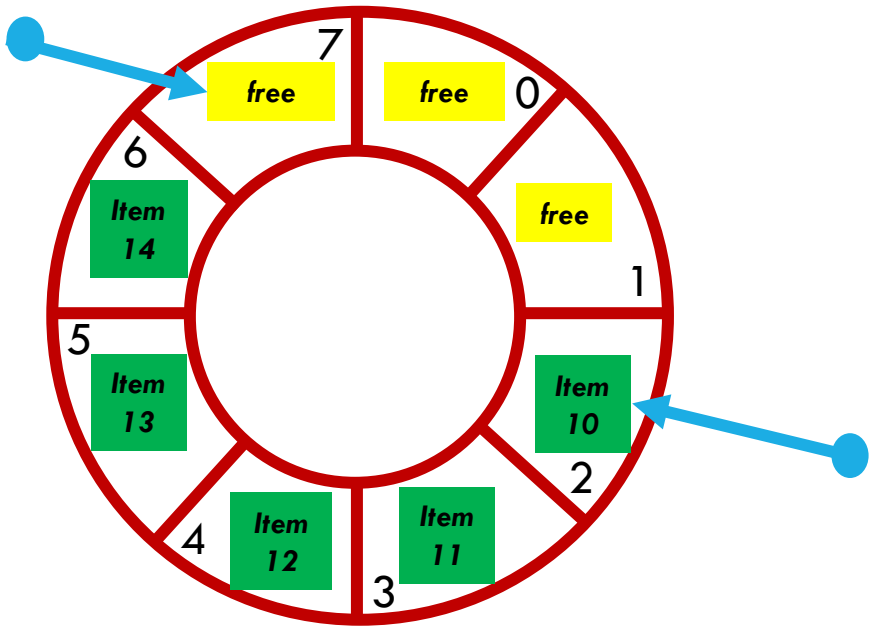
- An application is divided into stages
- One stage has one or more threads that "produce" some objects, like lines read from files.
- A second stage has one or more threads that "consume" this data, for example by counting words in those lines.

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

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"

BUT HOW 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

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

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.

TOOLKIT NEEDED

If multiple producers simultaneously try and produce an item, they would be accessing **nfree** and **free_ptr** simultaneously. Moreover, filling a slot will also increment **nfull.**

Producers also need to wait if **nfree == 0:** The buffer is full.

… and they will want fairness: no producer should get more turns than the others, if they are running concurrently.

A PRODUCER OR CONSUMER WAITS IF NEEDED

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

Producer:

```
void produce(const Foo& obj)
{
    std::scoped_lock plock(mtx);
    if(nfree == 0) wait;
    buffer[free_ptr++ % LEN] = obj;
    ++nfull;
    - - nfree;
}
```

Consumer:

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

DRILL DOWN…

It takes a moment to understand the race condition.

When we wait for some property to be true (some "condition"), we want to atomically enter a wait state and simultaneously release the monitor lock. This way we sure to get any notifications.

Releasing the lock before waiting could "miss" a notification.

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.

SOLUTION TO THE BOUNDED BUFFER PROBLEM USING A MONITOR PATTERN

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

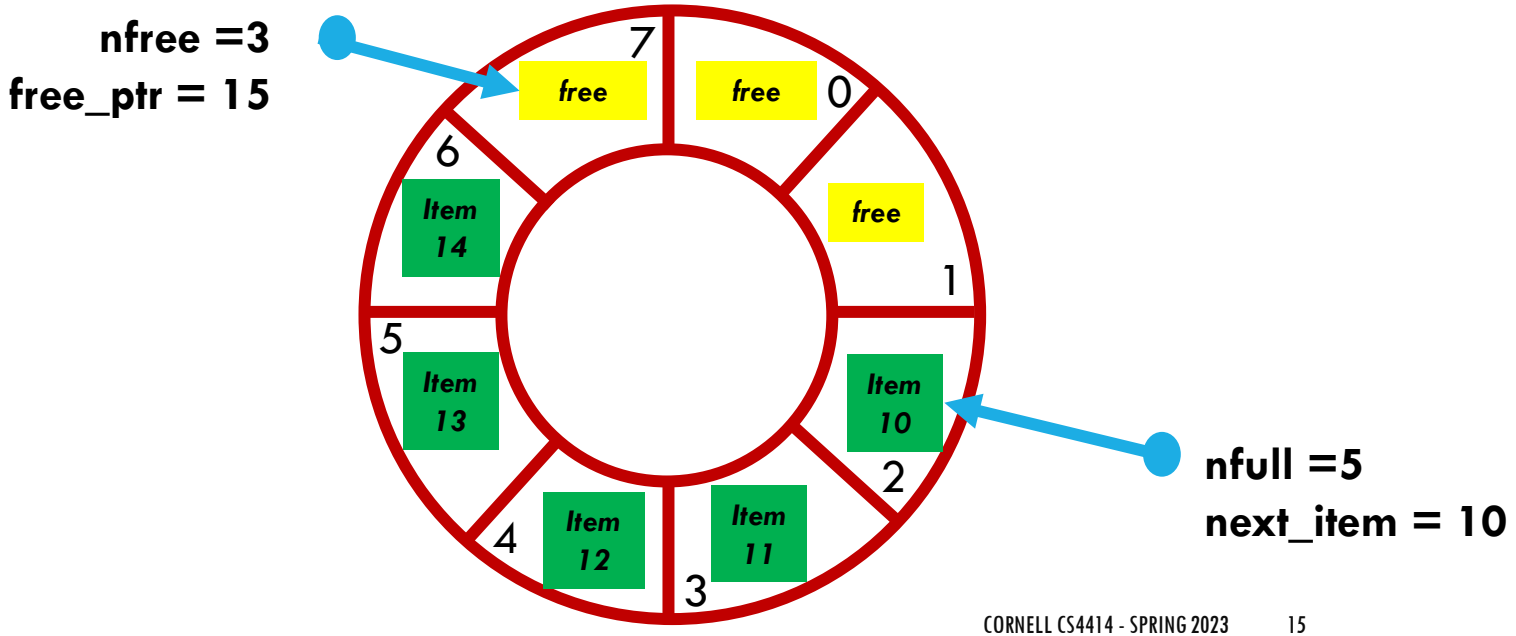
```
std::mutex mtx;
std::condition_variable not_empty;
std::condition_variable not_full;
```

… even though we will have two critical sections (one to produce, one to consume) we use one mutex.

SOLUTION TO THE BOUNDED BUFFER PROBLEM USING A MONITOR PATTERN

Next, 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;
```



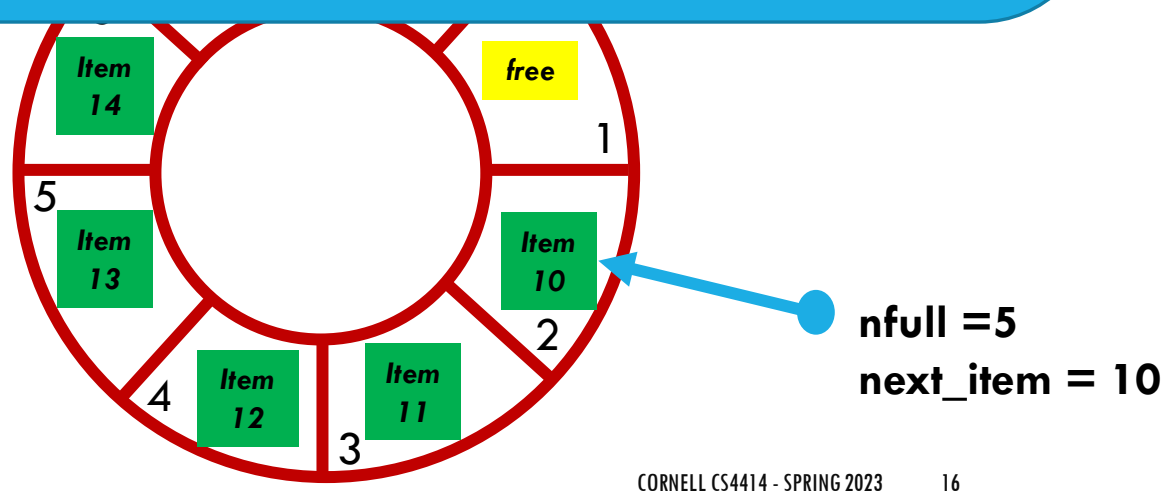
SOLUTION TO THE BOUNDED BUFFER PROBLEM USING A M

Next, we need our const int L
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);
        while(nfree == 0)
                not_full.wait(plock);
         buffer[free_ptr++ % LEN] = obj;
         --nfree;
         ++nfull;
         not_empty.notify_one();
}
```

CODE TO PRODUCE AN

```
void produce(const Foo&
{
        std::unique_lock plock(mtx);
        while(nfree == 0)
                not_full.wait(plock);
        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

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

A unique_lock is a lot like a scoped_lock but offers some extra functionality internally used by wait and notify

CODE TO PRODUCE AN

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

The while loop is needed because there could be multiple threads trying to produce items at the same time. Notify would wake *all* of them up, so we need the unlucky ones to go back to sleep!

CODE TO PRODUCE

```
void produce(const Foo& ob
{
        std::unique_lock pl
        while(nfree == 0)
            not_full.wait(plock);
        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);
        while(nfree == 0)
                not_full.wait(plock);
         buffer[free_ptr++ % LE
         --nfree;
         ++nfull;
         not_empty.notify_one();
}
```

We produced one item, so if multiple consumers are waiting, we just wake one of them up – no point in using notify_all

CODE TO CONSUME AN ITEM

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

CODE TO CONSUME A

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

Although the notify occurs before we read and return the item, the scoped-lock won't be released until the end of the block. Thus the return statement is still protected by the lock.

DID YOU NOTICE THE "WHILE" LOOPS?

A condition variable is used when some needed property does not currently hold. It allows a thread to wait.

In most cases, you can't assume that the property holds when your thread wakes up after a wait! *This is why we often recheck by doing the test again.*

This pattern protects against unexpected scheduling sequences.

CLEANER NOTATION, WITH A LAMBDA

We wrote out the two while loops, so that you would know they are required.

But C++ has a nicer packaging, using a lambda notation for the condition in the while loop.

CODE TO PRODUCE AN ITEM

```
void produce(const Foo& obj)
{
        std::unique_lock plock(mtx);
        while(nfree == 0)
              not_full.wait(plock);
         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

This means "capture all by reference". The lambda can access any locally scoped variables by reference.

```
void prod
{
        std::unique_lock plock(mtx);

        not_full.wait(plock, [&](){ return nfree != 0;});

        buffer[free_ptr++ % LEN] = obj;
        --nfree;
        ++nfull;
        not_empty.notify_one();
}
```

COD

The condition is "what you are waiting for", not "why you are waiting". So it is actually the negation of what would have been in the while loop!

```
void pr
{
        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 CONSUME AN ITEM

```
Foo consume()
{
        std::unique_lock clock(mtx);
        while(nfull == 0)
              not_empty.wait(clock);
         ++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(clock, [&]() { return nfull != 0; });

        ++nfree;
        --nfull;
        not_full.notify_one();
        return buffer[full_ptr++ % LEN];
}
```

A SECOND EXAMPLE

The “readers and writers” pattern captures this style of sharing for arrays, or for objects like std::list and std::map.

The key observation: 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 a form of fairness: reads should not *starve* updates

EXPRESED AS A MONITOR WITH WHILE LOOPS

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

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

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

… 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_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();
}
```

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 });
```

THIS VERSION 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). But a symmetric solution is very hard to design.

WARNING ABOUT "SPURIOUS WAKEUPS"

Older textbooks will show readers and writers using an "if" statement, not a while loop. But this is not safe with modern systems.

If you read closely, that old code assumed that a wait only wakes up in the event of a notify_one or notify_all. But such systems can hang easily if nobody does a notify – a common bug.

Modern condition variables *always* wake up after a small delay, even if the condition isn't true.

NOTIFY_ALL VERSUS NOTIFY_ONE

notify_all wakes up *every* waiting thread. We used it here.

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.

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.

BASICALLY, WE DON'T WORRY ABOUT FAIRNESS

Standard code focuses on safety (nothing bad will happen) and liveness (eventually, something good will happen).

Fairness is a wonderful concept but brings too much complexity.

So we trust in randomness to give us an adequate approximation to fairness.

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.

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;
```

… 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.

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.

REMARK: OLDER PATTERNS

C++ has evolved in this area, and has several templates for lack management. Unfortunately, they have duplicated functions

unique_lock -- very general, flexible, powerful. But use this only if you actually need all its features.

lock_guard -- a C++ 11 feature, but it turned out to be buggy in some situations. **Deprecated.**

scoped_lock -- C++ 17, can lock multiple mutex objects in one deadlock-free atomic action.

MONITOR SUMMARY

atomic<t> for base types (int, float, etc), volatile, test-and-set…

unique_lock and scoped_lock (C++ 17).

Monitor pattern: combines a mutex with condition variables to offer protection as well as a wait and notify mechanism, all integrated with locking in an atomic and safe way.