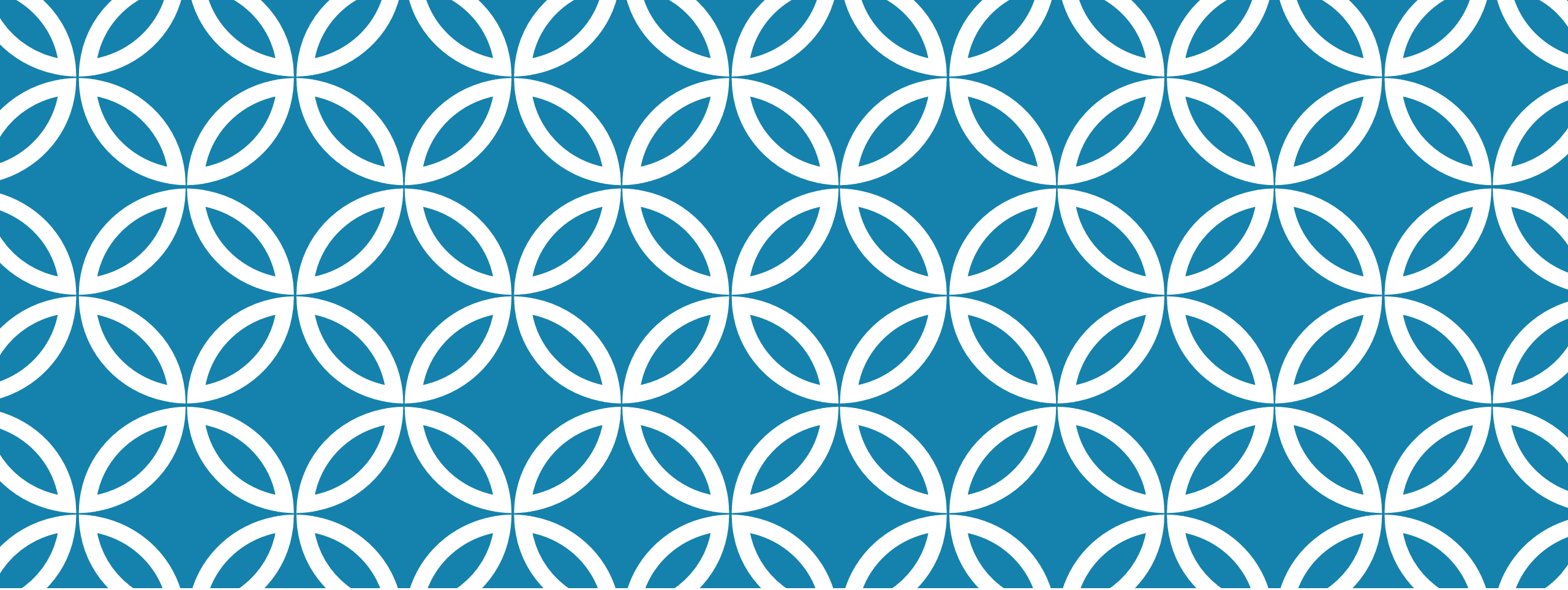


SYNCHRONIZATION PRIMITIVES

Professor Ken Birman

CS4414 Lecture 15

IDEA MAP FOR MULTIPLE LECTURES!

Reminder: Thread Concept

Lightweight vs. Heavyweight

Thread "context" and scheduling

C++ mutex objects. Atomic data types.

Race Conditions, Deadlocks, Livelocks

Today: Focus on the danger of sharing without synchronization and the hardware primitives we use to solve this.

IDEA MAP FOR TODAY'S LECTURE!

- Today: Focus on the danger of sharing without synchronization and the hardware/software primitives we can use to solve this.
- Issue we will be looking at: there are too many options, yet none of them really is "elegant" for reasoning about safety in a big program!
- Today we'll see many options. The recommended answer (monitors) will be in the lectures next week.

… WITH CONCURRENT THREADS, SOME SHARING IS USUALLY NECESSARY

Suppose that threads A and B are sharing an integer **counter**. What could go wrong?

We touched on this briefly in an early lecture. A and B both simultaneously try to increment **counter**. But an increment occurs in steps: load the variable (**counter**), add one, save it back.

… they conflict, and we "lose" one of the counting events.

THREADS A AND B SHARE A COUNTER

Thread A:

counter++;

```
movq    counter,%rax
addq    $1,%rax
movq    %rax,counter
```

Thread B:

counter++;

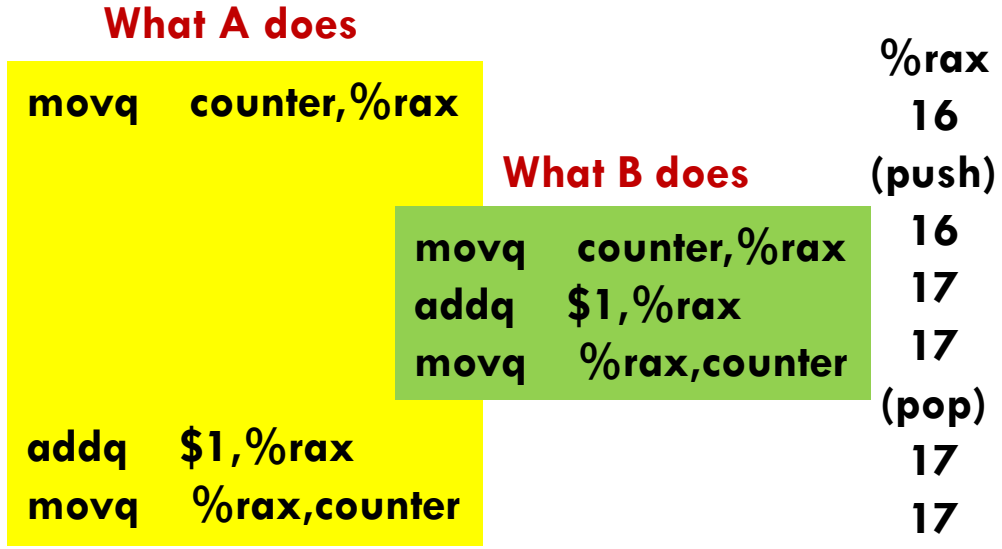
```
movq    counter,%rax
addq    $1,%rax
movq    %rax,counter
```

Either context switching or NUMA concurrency could cause these instruction sequences to interleave!

EXAMPLE: COUNTER IS INITIALLY 16, AND BOTH A AND B TRY TO INCREMENT IT.

The problem is that A and B have their own private copies of the counter in %rax

With pthreads, each has a private set of registers: a private %rax



With lightweight threads, context switching saved A's copy while B ran, but then reloaded A's context, which included %rax

THIS INTERLEAVING CAUSES A BUG!

If we increment 16 twice, the answer should be 18.

If the answer is shown as 17, all sorts of problems can result.

Worse, the schedule is unpredictable. This kind of bug could come and go…

STANDARD C++ LIBRARY GUARANTEE

Suppose you are using the C++ std library:

- No lock is needed when accessing *different* objects
- Methods of a single object can simultaneously be called by arbitrarily many **read-only** threads. No locks are needed.
- … But only a single active **writer** is permitted (this excludes other writers as well as new or already-active readers).

… You must protect against having multiple writers or a mix of readers and writers concurrently accessing the same object.

BOOST WARNING: READ DOCUMENTATION!

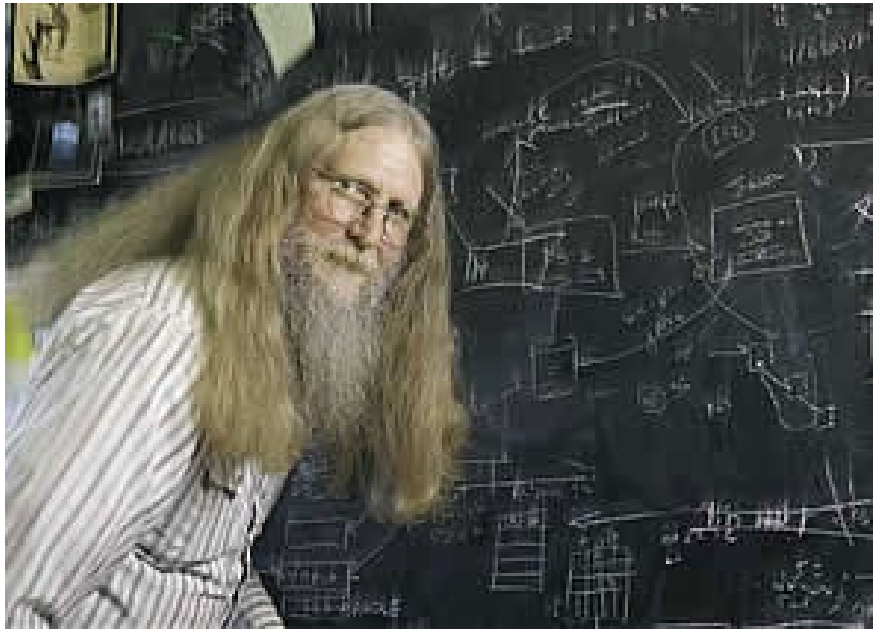
Some people love a library called Boost, which has many things lacking in C++ std:: today. But the guarantees vary for different Boost tools, which is one reason many companies are hesitant to use Boost

Some Boost libraries are "thread safe" meaning they implement their own locking. That would be *more* than what std:: promises.

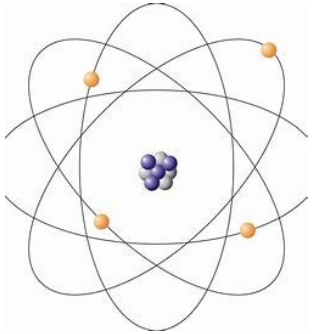
Some are like std::. And some just specify their own rules!

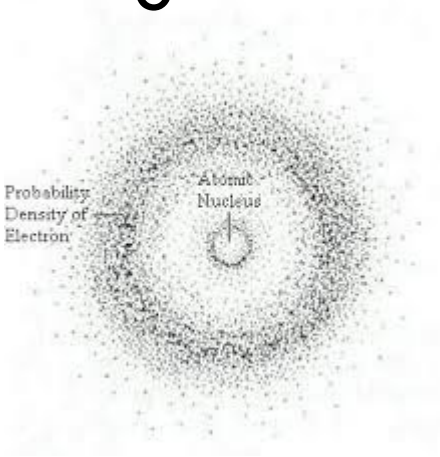
BRUCE LINDSAY

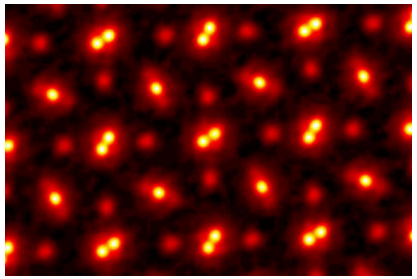
A famous database researcher



Bruce coined the terms “Bohrbugs” and “Heisenbugs”

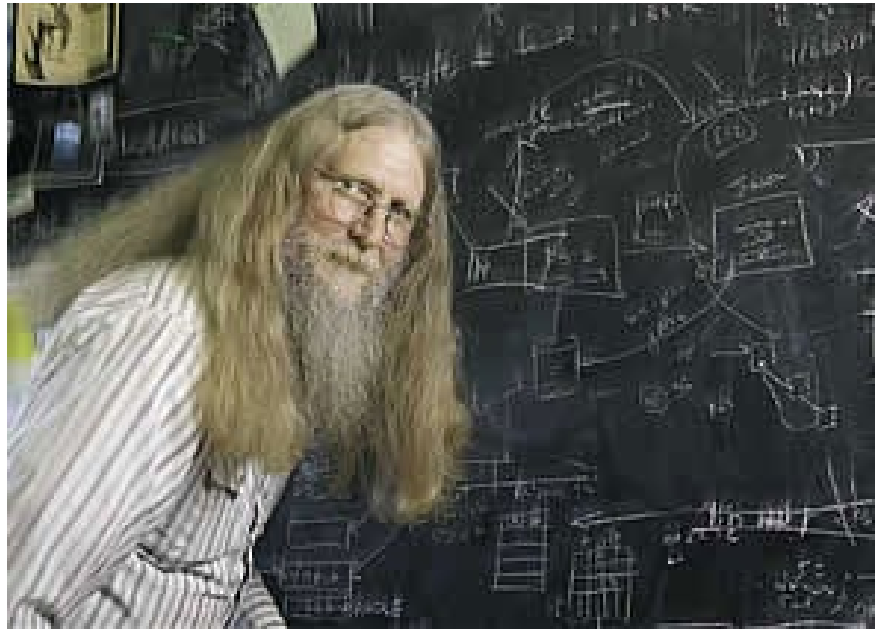






praseodymium orthoscandate ($PrScO_3$) crystal zoomed in 100 million times

BRUCE LINDSAY



In a concurrent system, we have two kinds of bugs to worry about

A Bohrbug is a well-defined, reproducible thing. We test and test, find it, and crush it.

Concurrency can cause Heisenbugs… they are very hard to reproduce. People often misunderstand them, and just make things worse and worse by patching their code without fixing the root cause!

CONCEPT: CRITICAL SECTION

A critical section is a block of code that accesses variables that are read **and updated.** You must have two or more threads, at least one of them doing an update (writing to a variable).

The block where A and B access the counter is a critical section. In this example, both update the counter.

Reading constants or other forms of unchanging data is not an issue. And you can safely have many simultaneous *readers*.

WE NEED TO ENSURE THAT A AND B CAN'T BOTH BE IN THE CRITICAL SECTION AT THE SAME TIME!

Basically, when A wants to increment **counter,** A goes into the critical section… and locks the door.

Then it can change the counter safely.

If B wants to access **counter,** it has to wait until A unlocks the door.

C++ ALLOWS US TO DO THIS.

```
std::mutex mtx;


void safe_inc(int& counter)
{
        std::scoped_lock  lock(mtx);
        counter++;
}
```

C++ ALLOWS US TO DO THIS.

```
std::mutex mtx;


void safe_inc(int& counter)
{
        std::scoped_lock  lock(mtx);
        counter++;                    // A critical section!
}
```

C++ ALLOWS US TO DO THIS.

```
std::mutex mtx;

void safe_inc(int& counter)
{
    std::scoped_lock lock(mtx);
    counter++;                  // A critical section!
}
```

This is a C++ type!

C++ ALLOWS US TO DO THIS.

```
std::mutex mtx;


void safe_inc(int& counter)
{
        std::scoped_lock lock(mtx);
        counter++;                    // A critical section!
}
```

This is a variable name!

C++ ALLOWS US TO DO THIS.

```
std::mutex mtx;


void safe_inc(int& counter)
{
        std::scoped_lock  lock(mtx);
        counter++;                    // A critical section!
}
```

The mutex is passed to the scoped_lock constructor

HOW DOES SCOPED_LOCK WORK?

```
boolean mutex = 0;
while(test_and_set(mutex) == 1)
                    /* just wait, looping */ ;
 … now I hold the lock!
 mutex = 0;     /* release the lock */
```

HOW DOES SCOPED_LOCK WORK?

In effect, keep trying to "be the winner" who sets mutex to 1.

- If "this try" flipped it from 0 to 1, my thread just got the lock
- If it was already 1, some other thread holds the lock.

The pattern is called **spinning** or **busy waiting**. As for the **test_and_set,** in fact Intel offers an "atomic" **test_and_set** instruction that will set a bit, but also test its old value. There is also one called **compare_and_swap.**

COMMON MISTAKE

Very easy to forget the variable name! This legal C++ yet not all all what you intended ☹

```
std::scoped_lock  lock(mtx);
```

If you make that mistake…

- C++ does run the constructor
- But then the new object immediately goes out of scope.
- Effect is to acquire but then instantly release the lock

MULTIPLE MUTEX VARIABLES

std::scoped_lock can acquire and release a list of mutex variables in a single atomic action

This is a better choice than acquiring them one by one because it cannot cause a deadlock, but we do not always know which locks we will acquire.

- Common tricky case: while holding a lock, call a method that also needs a lock. Will look closely at this in lecture 17.

RULE: SCOPED_LOCK

```
std::scoped_lock  lock(mtx);
```

Your thread might pause when this line is reached.

Suppose counter is accessed in two places?

… use std::scoped_lock something(mtx) in both, *with the same mutex*. ***"The mutex, not the variable name, determines which threads will be blocked".***

WHAT DO WE MEAN BY "AT TWO PLACES"?

Suppose **counter** is a global integer. But many .cpp files declare it as an extern and access it.

Is this one critical section or many?

- It feels like many critical sections (each such access is at risk)
- But we view them as a single critical section that is entered from many places. They must use the identical mutex.

RULE: SCOPED_LOCK

```
std::scoped_lock  lock(mtx);
```

When a thread "acquires" a lock on a mutex, it has sole control!

You have "locked the door". Until the current code block exits, you hold the lock and no other thread can acquire it!

Upon exiting the block, the lock is released (this works even if you exit in a strange way, like throwing an exception)

PEOPLE USED TO THINK LOCKS WERE THE SOLUTION TO ALL OUR CHALLENGES!

They would just put a std::scoped_lock whenever accessing a critical section.

They would be very careful to use the same mutex whenever they were trying to protect the same resource.

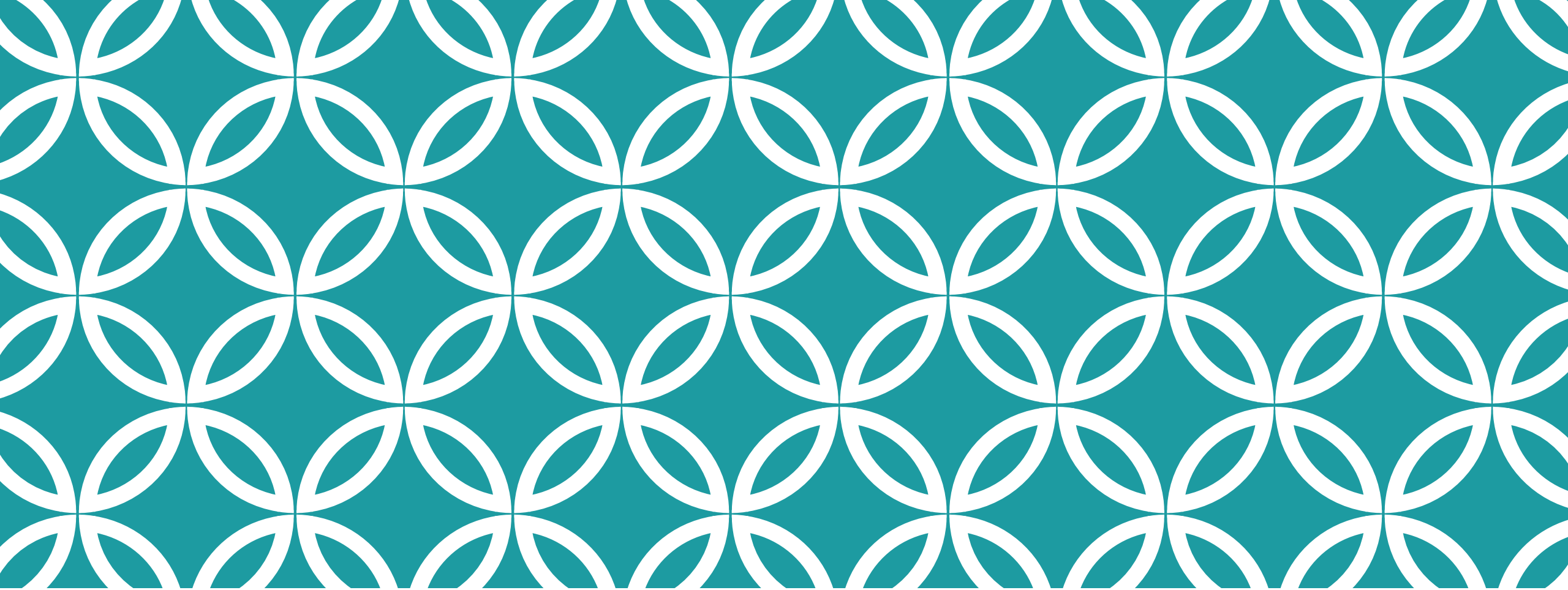
It felt like magic! At least, it did for a little while…

MORE ISSUES TO CONSIDER

Data structures: The thing we are accessing might not be just a single counter.

Threads could share a std::list or a std::map or some other structure with pointers in it. These complex objects may have a complex representation with several associated fields.

Moreover, with the alias features in C++, two variables can have different names, but refer to the same memory location.



NOW, A TOUR OF OUR OPTIONS

It isn't just mutex and scoped_lock!

HARDWARE ATOMICS

Hardware designers realized that programmers would need help, so the hardware itself offers some guarantees.

First, memory accesses are *cache line atomic*.

What does this mean?

CACHE LINE: A TERM WE HAVE SEEN BEFORE!

All of NUMA memory, including the L2 and L3 caches, are organized in blocks of (usually 64) bytes.

Such a block is called a cache line for historical reasons. Basically, the "line" is the width of a memory bus in the hardware.

CPUs load and store data in such a way that any object that fits in one cache line will be *sequentially consistent*.

SEQUENTIAL CONSISTENCY

Imagine a stream of reads and writes by different CPUs

Any given cache line sees a *sequence* of reads and writes. A read is guaranteed to see the value determined by the prior writes.

For example, a CPU never sees data "halfway" through being written, if the object lives entirely in one cache line.

SEQUENTIAL CONSISTENCY

Sequential consistency is a “read my own writes” policy

- A thread does some updates
- With modern memory, it can take time for those to reach the memory unit and for caches to become coherent
- Sequential consistency is the property that if thread A does some writes, then reads its own writes, it is guaranteed to see them in the order it issued them.

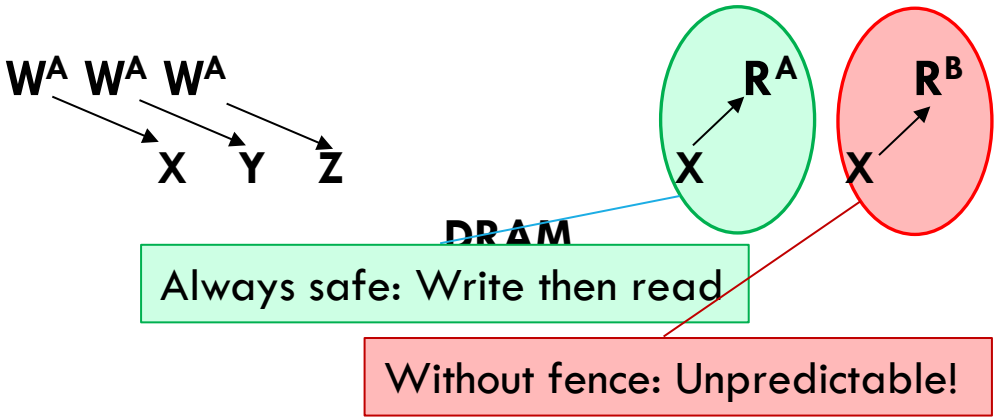
MEMORY FENCING

But suppose that A does writes and B (some other thread) does the reads. Will it see them?

A "memory fence" is a hardware feature that guarantees this. If A issues a memory fence, B is certain to see all of A's prior reads. A memory-fenced instruction needs a few nanoseconds to ensure this.

Locking uses an atomic with a built-in memory fenced behavior.

MEMORY FENCING

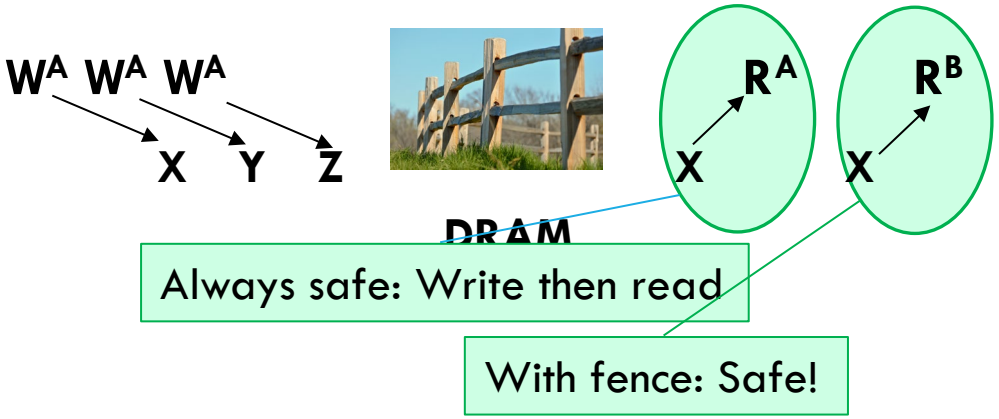


But suppose that A does writes and B (some other thread) does the reads. Will it see them?

A "memory fence" is a hardware feature that guarantees this. If A issues a memory fence, B is certain to see all of A's prior reads. A memory-fenced instruction needs a few nanoseconds to ensure this.

Locking uses an atomic with a built-in memory fenced behavior.

MEMORY FENCING



But suppose that A does writes and B (some other thread) does the reads. Will it see them?

A "memory fence" is a hardware feature that guarantees this. If A issues a memory fence, B is certain to see all of A's prior reads. A memory-fenced instruction needs a few nanoseconds to ensure this.

Locking uses an atomic with a built-in memory fenced behavior.

MEMORY FENCING PUZZLE

Suppose I have one main thread and it initializes some objects. But then it forks off a bunch of child threads. They read those same objects. Will they see the initialized data? Is locking needed?

- After all: in this case we have (1) multiple threads, (2) shared data, (3) a writer and (4) some readers
- And so, **yes**, we need a memory fence of some form!

Fortunately, std::thread() does some locking and that creates a fence.

TEST_AND_SET, COMPARE_AND_SWAP

These instructions are examples of C++ std library methods constructed using std::atomics

But there are *many* things you can do directly with atomics

TERM: "ATOMICITY"

This means "all or nothing". **test_and_set** is an atomic instruction

It refers to a complex operation that involves multiple steps, but in which no observer ever sees those steps in action.

We only see the system before or after the atomic action runs.

HOW POWERFUL IS SEQUENTIAL CONSISTENCY?

This was a famous puzzle in the early days of computing: do we really need special instructions?

Is sequential consistency enough on its own? There were many proposed algorithms… and some were incorrect!

Eventually, two examples emerged, with nice correctness proofs

DEKKER'S ALGORITHM FOR TWO PROCESSES

P0 and P1 can enter freely, but if both try at the same time, the "turn" variable allows first one to get in, then the other.

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

Note: You are not responsible for Dekker's algorithm, we show it just for completeness.

DECKER'S ALGORITHM WAS…

Fairly complicated, and not small (wouldn't fit on one slide in a font any normal person could read)

Elegant, but not trivial to reason about.

In CS4410 we develop proofs that algorithms like this are correct, and those proofs are not simple!

Note: You are not responsible for Dekker's algorithm, we show it just for completeness.

LESLIE LAMPORT

Lamport extended Decker's for many threads,
but also developed a simpler proof of correctness

He uses a visual story to explain his algorithm: a Bakery with a ticket dispenser

Tickets

Note: You are not responsible for the Bakery algorithm, we show it just for completeness.

LAMPORT'S BAKERY ALGORITHM FOR N THREADS

If no other thread is entering, any thread can enter

If two or more try at the same time, the ticket number is used.

Tie? The thread with the smaller id goes first

```
0   // declaration and initial values of global variables
1   Entering: array [1..NUM_THREADS] of bool = {false};
2   Number: array [1..NUM_THREADS] of integer = {0};
3
4   lock(integer i) {
5       Entering[i] = true;
6       Number[i] = 1 + max(Number[1], ..., Number[NUM_THREADS]);
7       Entering[i] = false;
8       for (integer j = 1; j <= NUM_THREADS; j++) {
9           // Wait until thread j receives its number:
10          while (Entering[j]) { /* nothing */ }
11          // Wait until all threads with smaller numbers or with the same
12          // number, but with higher priority, finish their work:
13          while ((Number[j] != 0) && ((Number[j], j) < (Number[i], i))) { /* nothing */ }
14      }
15  }
16
17  unlock(integer i) {
18      Number[i] = 0;
19  }
20
21  Thread(integer i) {
22      while (true) {
23          lock(i);
24          // The critical section goes here...
25          unlock(i);
26          // non-critical section...
27      }
28  }
```

Note: You are not responsible for the Bakery algorithm, we show it just for completeness.

LAMPORT'S CORRECTNESS GOALS

An algorithm is *safe* if "nothing bad can happen." For these mutual exclusion algorithms, safety means "at most one thread can be in a critical section at a time."

An algorithm is *live* if "something good eventually happens". So, eventually, some thread is able to enter the critical section.

An algorithm is *fair* if "every thread has equal probability of entry"

Note: You are not responsible for the Bakery algorithm, we show it just for completeness.

THE BAKERY ALGORITHM IS TOTALLY CORRECT

It can be proved safe, live and even fair.

For many years, this algorithm was actually used to implement locks, like the scoped_lock we saw on slide 11

These days, the C++ libraries for synchronization use **atomics**, and we use the library methods (as we will see in Lecture 15).

Note: You are not responsible for the Bakery algorithm, we show it just for completeness.

IMPLICATION?

Once we have sequential consistency, we can create any form of atomic object we like!

So in this perspective, using locking to safely update a list or a vector is just an "instance" of a more general capability!

Solutions like std::scoped_lock are higher level abstractions that can be built from lower-level sequentially consistent memory, perhaps with help from hardware instructions like **test_and_set**

ATOMIC MEMORY OBJECTS

Modern hardware supports atomicity for memory operations.

If a variable is declared to be atomic, using the C++ atomics templates, then basic operations occur to completion in an indivisible manner, even with NUMA concurrency.

For example, we could just declare

```
std::atomic<int>  counter;          // Now ++ is thread-safe
```

C / C++ ATOMICS

They actually come in many kinds, with slightly different properties built in

So-called **weak atomics** // FIFO updates, might "see" stale values
Acquire-release atomics // Like a "memory fence" (slide 35)
Stong atomics // Like using a mutex lock

SOME ISSUES WITH ATOMICS

The strongest atomics (mutex locks) are slow to access: we wouldn't want to use this annotation frequently!

The weaker forms are cheap but very tricky to use correctly

Often, a critical section would guard multiple operations. With atomics, the *individual* operations are safe, but perhaps not the block of operations.

VOLATILE

Volatile tells the compiler that a non-atomic variable might be updated by multiple threads… the value could change at any time.

This prevents C++ from caching the variable in a register as part of an optimization. *But the hardware itself could still do caching.*

Volatile is needed if you do completely unprotected sharing. With C++ library synchronization, you never need this keyword.

WHEN WOULD YOU USE VOLATILE?

Suppose that thread A will do some task, then set a flag "A_Done" to true. Thread B will "busy wait":

```
while(A_Done == false) ;          // Wait until A is done
```

Here, we need to add **volatile** (or **atomic**) to the declaration of A_Done. Volatile is faster than atomic, which is faster than a lock.

HIGHER LEVEL SYNCHRONIZATION: BINARY AND COUNTING SEMAPHORES (~1970'S)

We'll discuss the counting form

- A form of object that holds a lock and a counter. The developer initializes the counter to some non-negative value.
- **Acquire** pauses until counter > 0, then decrements counter and returns
- **Release** increments semaphore (if a process is waiting, it wakes up).

C++ has semaphores. The pattern is easy to implement.

Semaphores are a big topic in CS4410, but in CS4414 we don't use them because monitors are more powerful and easier to reason about.

PROBLEMS WITH SEMAPHORES

It turned out that semaphores were a cause of many bugs. Consider this code that protects a critical section:

```
mySem.acquire();
do something;                // This is the critical section
mySem.release();
```

… unusual control flow could prevent the release(), such as a **return** or **continue** statement, or a **caught exception**.

Semaphores are a big topic in CS4410, but in CS4414 we don't use them because monitors are more powerful and easier to reason about.

PROBLEMS WITH SEMAPHORES

It is also tempting to use semaphores as a form of "go to"

Process A

runB.release();

Process B

runB.acquire();

This is kind of ugly and can easily cause confusion

Semaphores are a big topic in CS4410, but in CS4414 we don't use them because monitors are more powerful and easier to reason about.

BETTER HIGH-LEVEL SYNCHRONIZATION

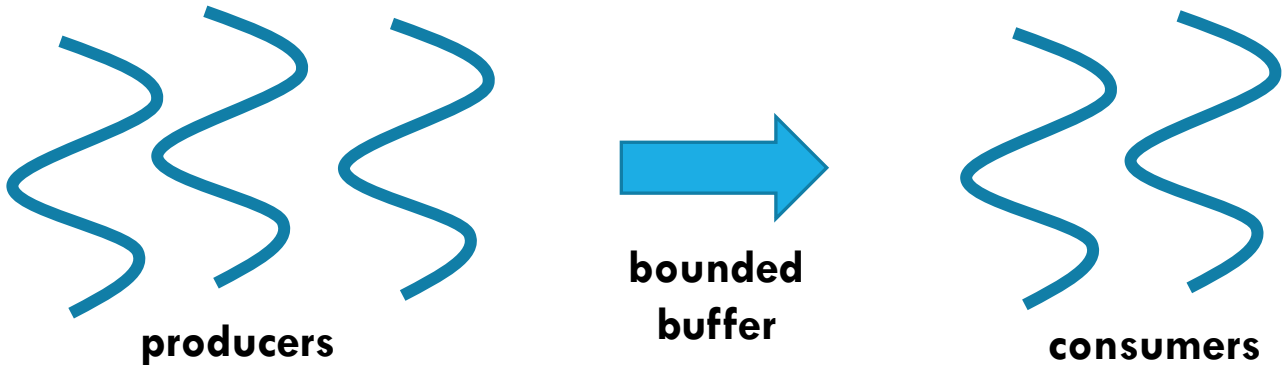
The complexity of these mechanisms led people to realize that we need higher-level approaches to synchronization that are safe, live, fair and make it easy to create correct solutions.

Let's look at an example of a higher level construct: a bounded buffer

WHO NEEDS THEM?

We saw a way to pass a "task id" to a thread, in lecture 14. But sometimes we don't yet have the information we want to pass to child threads at launch time and must do it at runtime.

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



BOUNDED BUFFER (LIKE A LINUX PIPE!)

We have a set of threads.

Some produce objects (perhaps, cupcakes!)

Others consume objects (perhaps, children!)

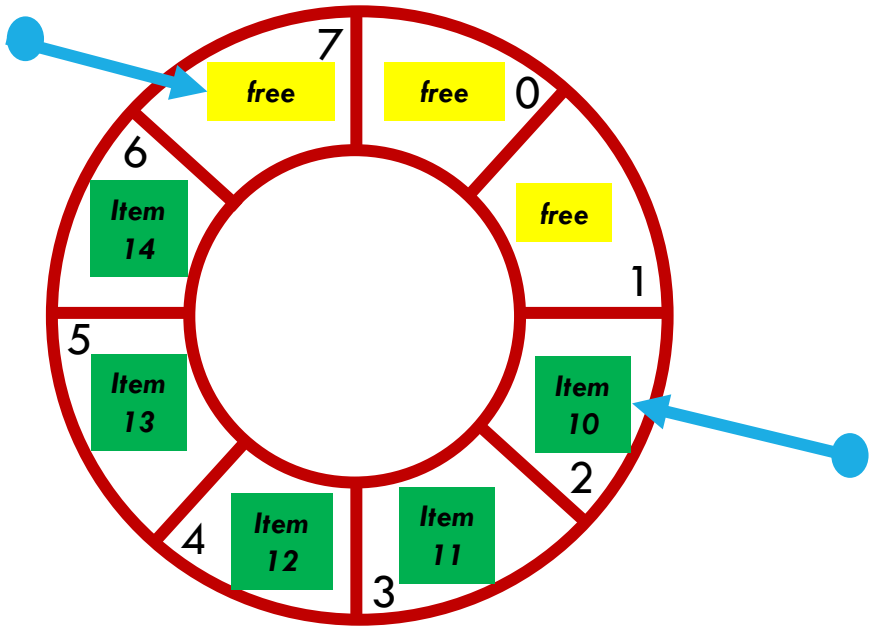
Goal is to synchronize the two groups.

A RING BUFFER

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 next free entry

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… and we can't fix it just by adding atomics or locks!

WE WILL SOLVE THIS PROBLEM IN LECTURE 16

Doing so yields a very useful primitive!

Putting a safe bounded buffer between a set of threads is a very effective synchronization pattern!

Example: In fast-wc we wanted to open files in one thread and scan them in other threads. A bounded buffer of file objects ready to be scanned was a perfect match to the need!

WHY ARE BOUNDED BUFFERS SO HELPFUL?

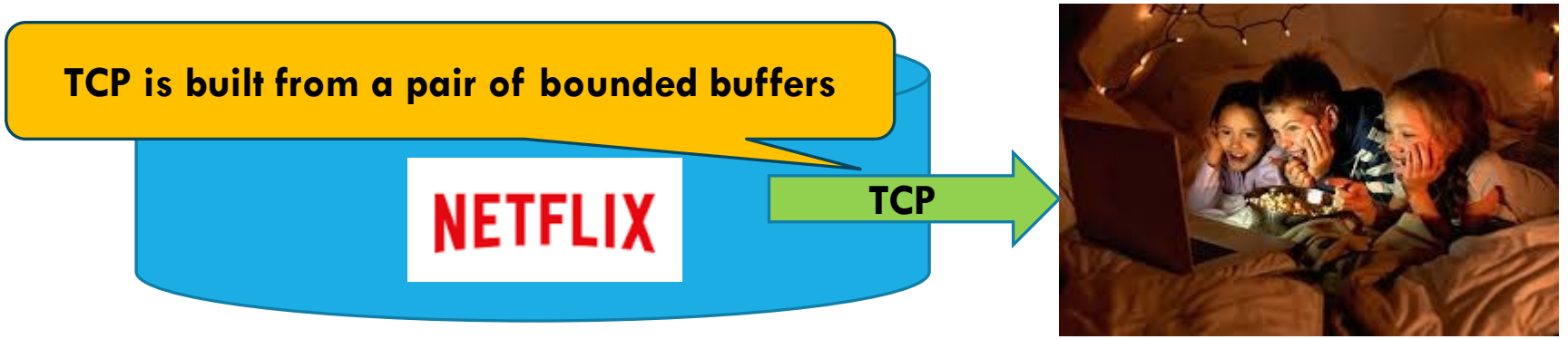
… in part, because they are safe with concurrency.

But they also are a way to absorb *transient rate mismatches*.

- A baker prepares batches of 24 cupcakes at a time.
- The school children buy them one by one.

If LEN ≥ 24, a bounded buffer of LEN cupcakes lets our baker make new batches *continuously*. The children can snack whenever they like.

TCP



The famous TCP networking protocol builds a bounded buffer that has two replicas separated by an Internet ink.

On one side, we have a server (perhaps, streaming a movie).

On the other, a consumer (perhaps, showing the movie)!

BUT ONE SIZE DOESN'T "FIT ALL CASES"

Only some use cases match this bounded buffer example (which, in any case, we still need to solve!)

Locks, similarly, are just a partial story.

So we need to learn to do synchronization in complex situations.

CRITICAL SECTIONS CAN BE SUBTLE!

By now we have seen several forms of aliasing in C++, where a variable in one scope can also be accessed in some other scope, perhaps under a different name.

In C++ it is common to overload operators like +, -, even [ ]. So almost any code could actually be calling methods in classes, or functions elsewhere in the program.

SUMMARY

Unprotected critical sections cause *serious* bugs!

Locks are an example of a way to protect a critical section, but the bounded buffer clearly needs "more"

What we really are looking for is a methodology for writing thread-safe code that uses C++ libraries safely.