

# ECE 2400 Computer Systems Programming

## Fall 2021

### Topic 16: Concurrent Programming

School of Electrical and Computer Engineering  
Cornell University

revision: 2021-08-28-21-44

1	C++ Threads	3
2	C++ Atomics	10
3	Drawing Framework Case Study	17

**zyBooks** The zyBooks logo is used to indicate additional material included in the course zyBook which will not be discussed in detail in lecture. Students are responsible for all material covered in lecture and in the course zyBook.

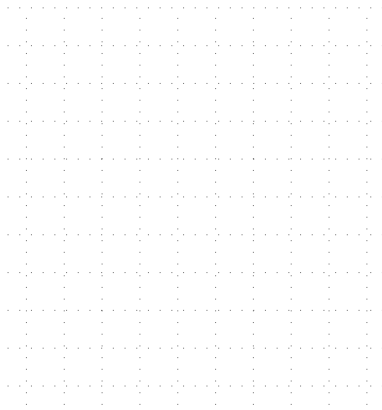
Copyright © 2021 Christopher Batten. All rights reserved. This handout was prepared by Prof. Christopher Batten at Cornell University for ECE 2400 / ENGRD 2140 Computer Systems Programming. Download and use of this handout is permitted for individual educational non-commercial purposes only. Redistribution either in part or in whole via both commercial or non-commercial means requires written permission.

- 
- Programming is organized around computations that execute *concurrently* (i.e., computations execute overlapped in time) instead of *sequentially* (i.e., computations execute one at a time)

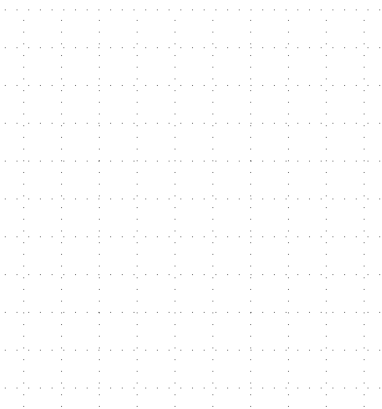
### **Processing an Array**



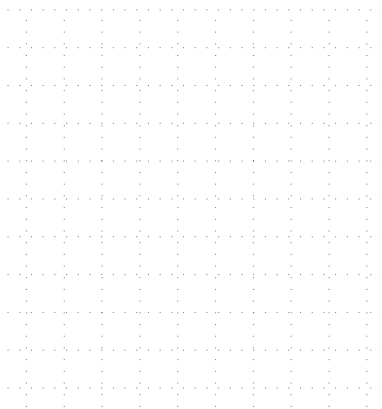
### **Sorting an Array**



### **Graphical User Interface**



### **Database Transactions**



## 1. C++ Threads

- Use object-oriented, generic, and functional programming to implement **threads**
  - Every thread has its own **independent stack and execution arrow**
  - Threads can access each other's variables through pointers or references
  - `std::thread` is a class provided by the C++ standard library
  - `std::thread` objects created using function pointers, functors, or lambdas
  - The pointer to the thread's stack will be its primary member field

```
1 #include <thread>
2
3 class thread
4 {
5     public:
6
7     template < typename Func, typename Arg0 >
8     thread( Func f, Arg0 a0 )
9     {
10         // create new stack
11         // set sp to point to new stack
12         // start executing function f(a0) using new stack
13         // return without waiting for function f to finish
14     }
15
16     void join()
17     {
18         // return when function f is finished
19     }
20
21     private:
22     stack_ptr_t sp;
23
24 };
```

```
main
t  thread
000 000 01 #include <thread>
000 000 02
000 000 03 void incr( int* x_p )
000 000 04 {
000 000 05     int y = *x_p;
000 000 06     int z = y + 1;
000 000 07     *x_p = z;
000 000 08 }
000 000 09
000 000 10 int main( void )
000 000 11 {
000 000 12     int a = 0;
000 000 13
000 000 14     std::thread t( &incr, &a );
000 000 15
000 000 16     t.join();
000 000 17     return 0;
000 000 18 }
```

stack

stack

- Use C++ functor to create a thread

```
1  class Incr
2  {
3  public:
4
5      Incr( int* x_p )
6          : m_x_p( x_p )
7      { }
8
9      void operator()() const
10     {
11         int y = *m_x_p;
12         int z = y + 1;
13         *m_x_p = z;
14     }
15
16     private:
17         int* m_x_p;
18 };
19
20 int main( void )
21 {
22     int a = 0;
23
24     Incr incr(&a);
25     std::thread t( &incr );
26
27     t.join();
28     return 0;
29 }
```

- Use C++ lambda to create a thread

```
1  int main( void )
2  {
3      int a = 0;
4
5      std::thread t( [&]()
6      {
7          int y = a;
8          int z = y + 1;
9          a = z;
10     });
11
12     t.join();
13     return 0;
14 }
```

```
main
t thread
000 000 01 #include <thread>
000 000 02
000 000 03 void avg( int* z_p, int x, int y )
000 000 04 {
000 000 05     int sum = x + y;
000 000 06     *z_p = sum / 2;
000 000 07 }
000 000 08
000 000 09 int main( void )
000 000 10 {
000 000 11     int a;
000 000 12     std::thread t( &avg, &a, 5, 10 );
000 000 13
000 000 14     int b;
000 000 15     avg( &b, 10, 15 );
000 000 16
000 000 17     t.join();
000 000 18     return 0;
000 000 19 }
```

stack

stack

## Parallel Vector-Vector Add

```
1  #include <thread>
2
3  void vvadd( int* dest, int* src0, int* src1,
4             int lo, int hi )
5  {
6      for ( int i = lo; i < hi; i++ )
7          dest[i] = src0[i] + src1[i];
8  }
9
10 int main( void )
11 {
12     const int size = N;
13     int src0[size] = { ... };
14     int src1[size] = { ... };
15     int dest[size];
16
17     int middle = size/2;
18     std::thread t( &vvadd, dest, src0, src1, 0, middle );
19
20     vvadd( dest, src0, src1, middle, size );
21
22     t.join();
23     return 0;
24 }
```

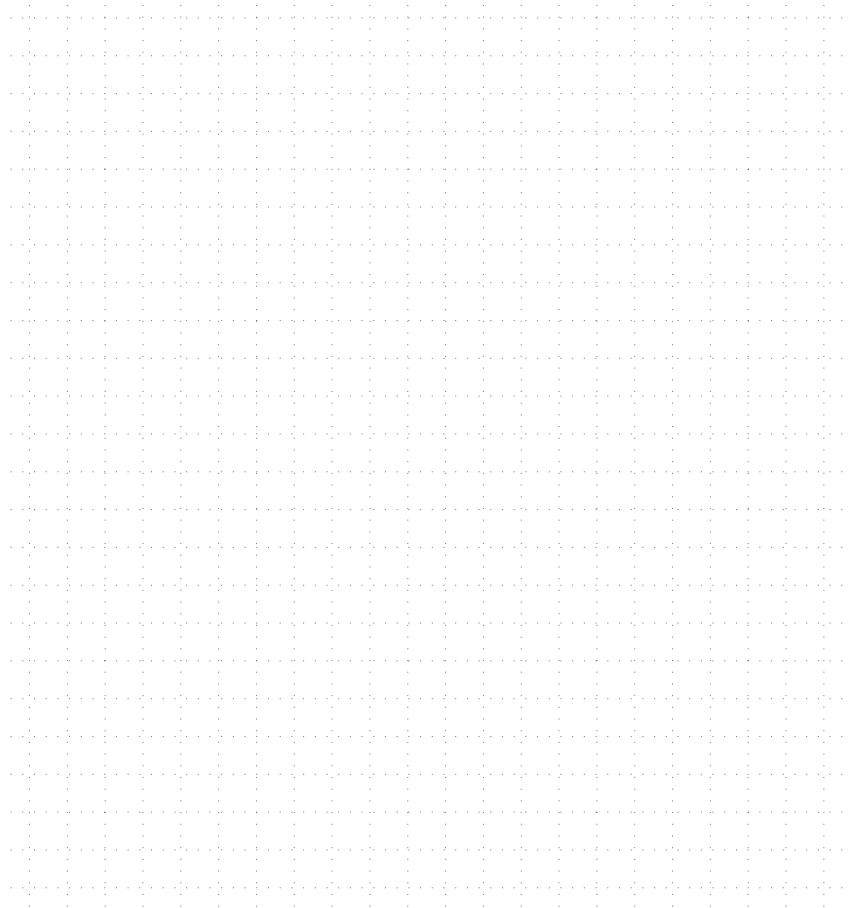
## Parallel Count Zeros

```
1  #include <thread>
2
3  void count_zeros( int* x, int* y, int begin, int end )
4  {
5      int count = 0;
6      for ( int i = begin; i < end; i++ )
7          if ( y[i] == 0 )
8              count++;
9      *x = count;
10 }
11
12 int main( void )
13 {
14     const int size = N;
15     int a[size] = { ... };
16
17     int mid1 = 1*(size/4);
18     int mid2 = 2*(size/4);
19     int mid3 = 3*(size/4);
20
21     // Array to store the results from each partition
22     int b[] = { 0, 0, 0, 0 };
23
24     // Count zeros in each partition in parallel
25     std::thread t0( &count_zeros, &b[0], a, 0, mid1 );
26     std::thread t1( &count_zeros, &b[1], a, mid1, mid2 );
27     std::thread t2( &count_zeros, &b[2], a, mid2, mid3 );
28     count_zeros( &b[3], a, mid3, size );
29
30     // Wait for all threads to finish
31     t0.join();
32     t1.join();
33     t2.join();
34
35     // Serial reduction
36     int c = 0;
37     for ( int i = 0; i < 4; i++ )
38         c += b[i];
39
40     return 0;
41 }
```



## Complexity Analysis

What is the execution time and time complexity as a function of  $N$  (size of array) with  $P$  (number of processors) as a key constant parameter?



## 2. C++ Atomics

- What if two threads increment the same variable?

```

main
t  thread
0000 0000 01  #include <thread>
0000 0000 02
0000 0000 03  void incr( int* x_p )
0000 0000 04  {
0000 0000 05      int y = *x_p;
0000 0000 06      int z = y + 1;
0000 0000 07      *x_p = z;
0000 0000 08  }
0000 0000 09
0000 0000 10  int main( void )
0000 0000 11  {
0000 0000 12      int a = 0;
0000 0000 13      std::thread t( &incr, &a );
0000 0000 14
0000 0000 15      incr( &a );
0000 0000 16
0000 0000 17      t.join();
0000 0000 18      return 0;
0000 0000 19  }

```

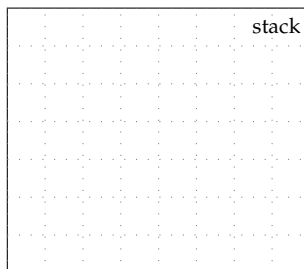
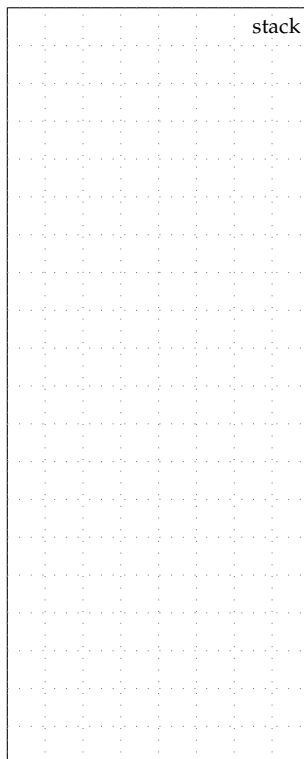
- Is a single C++ statement atomic?

```

1  void incr( int* x_p )
2  {
3      (*x_p)++;
4  }

```

<https://godbolt.org/g/zXLFXE>



## Using atomic operations

- The C++ standard library provides a templated atomic class which can enable atomic operations on various primitive types

```
1  #include <atomic>
2
3  template <>
4  class atomic<int>
5  {
6  public:
7      // constructors
8      atomic( int );
9
10     // overloaded operators
11     int operator++(int);
12     int operator++();
13     int operator--(int);
14     int operator--();
15     int operator+=( int v );
16     int operator-=( int v );
17     int operator&=( int v );
18     int operator|=( int v );
19     int operator^=( int v );
20
21     // atomic operations
22     int fetch_add( int v );
23     int fetch_sub( int v );
24     int fetch_and( int v );
25     int fetch_or ( int v );
26     int fetch_xor( int v );
27     ...
28
29 private:
30     int m_data;
31 }
```

```
1  #include <atomic>
2
3  template <>
4  atomic<int>::atomic( int v )
5  {
6      m_data = v;
7  }
8
9  // pseudo-code, must use special
10 // hardware instructions to
11 // guarantee all member functions
12 // are executed atomically!
13
14 template <>
15 int atomic<int>::operator++(int)
16 {
17     int prev = m_data;
18     m_data = m_data + 1;
19     return prev;
20 }
21
22 template <>
23 int atomic<int>::fetch_or( int v )
24 {
25     int prev = m_data;
26     m_data = m_data | v;
27     return prev;
28 }
```

```
1 #include <thread>
2 #include <atomic>
3
4 void incr( std::atomic<int>* x_p )
5 {
6     (*x_p)++; // guaranteed to execute atomically
7 }
8
9 int main( void )
10 {
11     std::atomic<int> a(0);
12     std::thread t( &incr, &a );
13
14     incr( &a );
15
16     t.join();
17     return 0;
18 }
```

<https://godbolt.org/g/bBeRzh>

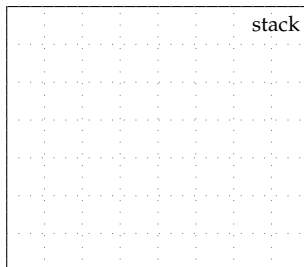
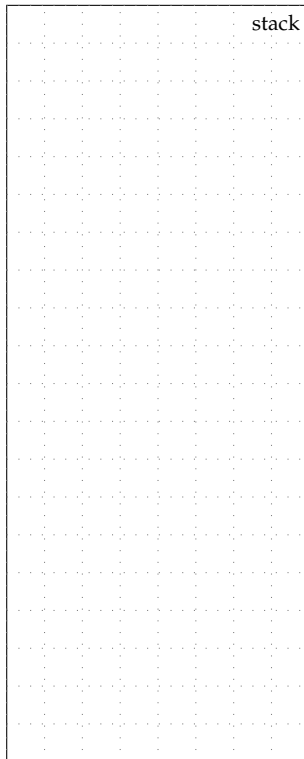
- What if we want to do something more complicated than this?
- How can we ensure a general piece of code is executed atomically?

- Use a **lock** to **guard** a **critical section**
  - exactly one thread can have the lock
  - use atomic operation to manipulate lock
  - 1. thread tries to acquire lock
  - 2. once acquired, execute critical section
  - 3. thread releases lock

```

main
t  thread
0000 0000 01 void incr( int* x_p,
0000 0000 02         std::atomic<int>* y_p )
0000 0000 03 {
0000 0000 04     // acquire lock
0000 0000 05     while ( y_p->fetch_or(1) == 1 )
0000 0000 06     { }
0000 0000 07
0000 0000 08     *x_p = foo(*x_p);
0000 0000 09
0000 0000 10     // release lock
0000 0000 11     *y_p = 0;
0000 0000 12 }
0000 0000 13
0000 0000 14 int main( void )
0000 0000 15 {
0000 0000 16     int a = 0;
0000 0000 17     std::atomic<int> b(0);
0000 0000 18
0000 0000 19     std::thread t( &incr, &a, &b );
0000 0000 20
0000 0000 21     incr( &a, &b );
0000 0000 22
0000 0000 23     t.join();
0000 0000 24     return 0;
0000 0000 25 }

```



## Encapsulate lock into a mutex

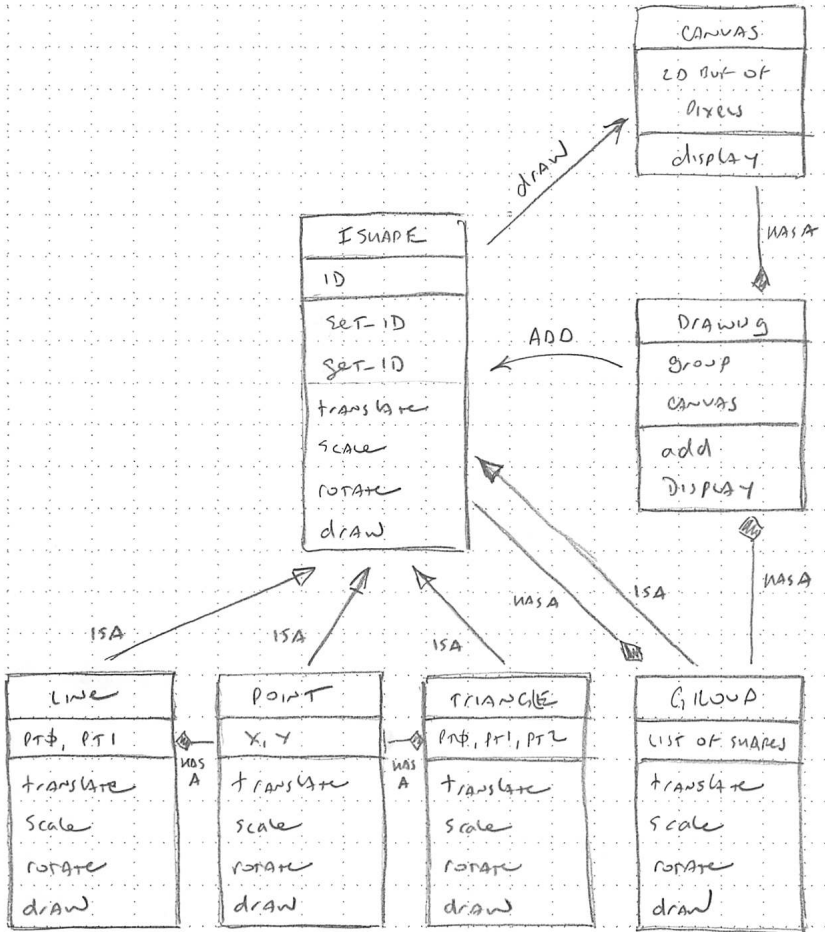
```
1  class Mutex
2  {
3  public:
4      Mutex() { m_lock = 0; }
5
6      void lock()
7      {
8          while ( m_lock.fetch_or(1) == 1 ) { }
9      }
10
11     void unlock() { m_lock = 0; }
12 private:
13     std::atomic<int> m_lock;
14 };
15
16 void incr( int* x_p, Mutex* m_p )
17 {
18     m_p->lock();
19
20     *x_p = foo(*x_p);
21
22     m_p->unlock();
23 }
24
25 int main( void )
26 {
27     int a = 0;
28     Mutex m;
29     std::thread t( &incr, &a, &m );
30     incr( &a, &m );
31     t.join();
32     return 0;
33 }
```

## RAII: Resource Acquisition Is Initialization

- What if we forget to unlock mutex? What if there is an exception?
- RAII is a design pattern that ties a resource to object lifetime (also known as **scope-bound resource management**)
- Acquire lock in constructor and release lock in destructor
- Elegantly ensures unlock is called for every lock even if an exception is thrown

```
1  class LockGuard
2  {
3      public:
4
5          LockGuard( Mutex* m )
6          {
7              m_mutex_p = m;
8              m_mutex_p->lock();
9          }
10
11         ~LockGuard()
12         {
13             m_mutex_p->unlock();
14         }
15
16         private:
17             Mutex* m_mutex_p;
18 };
19
20 void incr( int* x_p, Mutex* m_p )
21 {
22     LockGuard guard(m_p);
23     *x_p = foo(*x_p);
24 }
```

### 3. Drawing Framework Case Study





- Use concurrent programming to accelerate drawing many shapes

```
1  class Group : public IShape
2  {
3      public:
4          ...
5
6      void draw( Canvas* canvas ) const
7      {
8          assert( canvas != NULL );
9
10         // Use serial version if fewer than 1000 shapes
11
12         if ( m_shapes_size < 1000 ) {
13             for ( int i = 0; i < m_shapes_size; i++ )
14                 m_shapes[i]->draw( canvas );
15         }
16
17         // Use parallel version if 1000 or more shapes
18
19         else {
20             int middle = m_shapes_size/2;
21
22             // Child thread draws first half of shapes
23             std::thread t( [&]() {
24                 for ( int i = 0; i < middle; i++ )
25                     m_shapes[i]->draw( canvas );
26             } );
27
28             // Parent thread draws second half of shapes
29             for ( int i = middle; i < m_shapes_size; i++ )
30                 m_shapes[i]->draw( canvas );
31
32             t.join();
33         }
34     }
35 }
```