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

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.

- 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

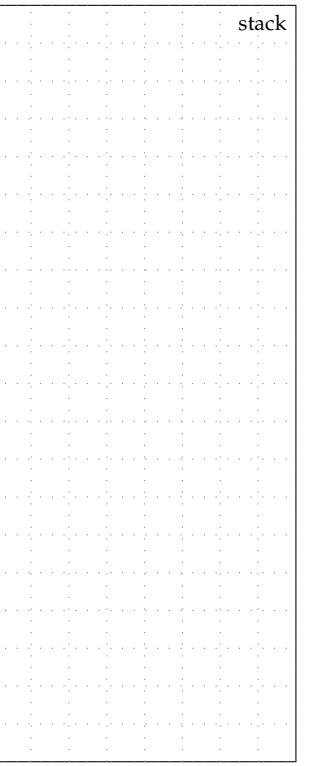
Database Transactions

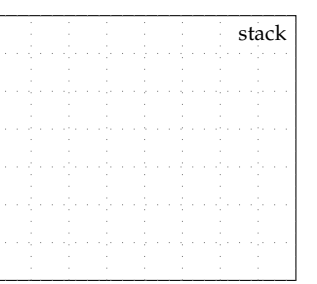
1. C++ Threads

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





- Use C++ functor to create a thread

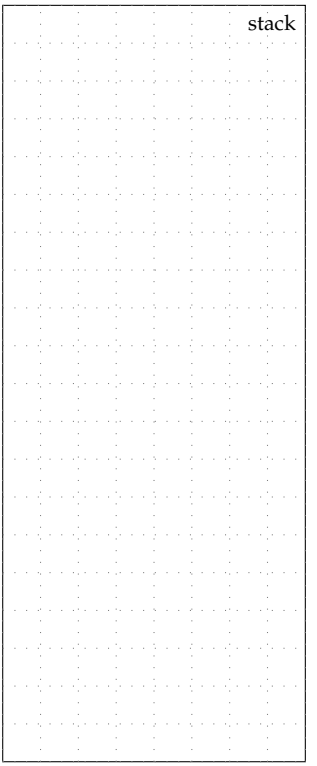
```
class Incr
{
 public:

  Incr( int* x_p )
   : m_x_p( x_p )
  { }

  void operator()() const
  {
    int y = *m_x_p;
    int z = y + 1;
    *m_x_p = z;
  }

 private:
  int* m_x_p;
};

int main( void )
{
  int a = 0;

  Incr incr(&a);
  std::thread t( &incr );

  t.join();
  return 0;
}
```

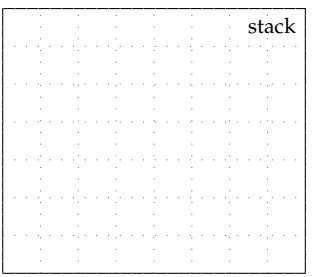
- Use C++ lambda to create a thread

```
int main( void )
{
  int a = 0;

  std::thread t( [&]()
  {
    int y = a;
    int z = y + 1;
    a = z;
  });

  t.join();
  return 0;
}
```

main
t thread

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





Parallel Vector-Vector Add

```
#include <thread>

void vvadd( int* dest, int* src0, int* src1,
            int lo, int hi )
{
  for ( int i = lo; i < hi; i++ )
    dest[i] = src0[i] + src1[i];
}

int main( void )
{
  const int size = N;
  int src0[size] = { ... };
  int src1[size] = { ... };
  int dest[size];

  int middle = size/2;
  std::thread t( &vvadd, dest, src0, src1, 0, middle );

  vvadd( dest, src0, src1, middle, size );

  t.join();
  return 0;
}
```

Parallel Count Zeros

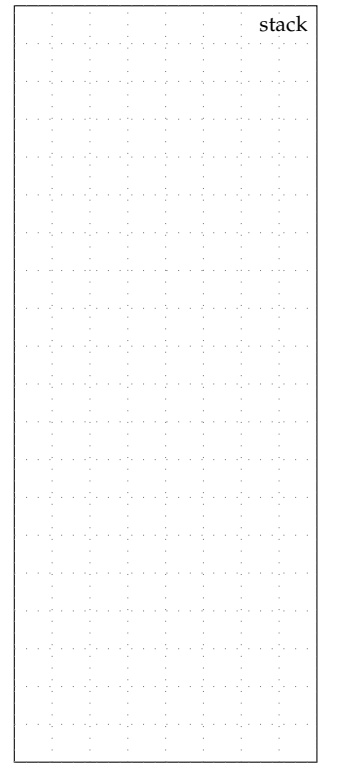
```
#include <thread>

void count_zeros( int* x, int* y, int begin, int end )
{
  int count = 0;
  for ( int i = begin; i < end; i++ )
    if ( y[i] == 0 )
      count++;
  *x = count;
}

int main( void )
{
  const int size = N;
  int a[size] = { ... };

  int mid1 = 1*(size/4);
  int mid2 = 2*(size/4);
  int mid3 = 3*(size/4);

  // Array to store the results from each partition
  int b[] = { 0, 0, 0, 0 };

  // Count zeros in each partition in parallel
  std::thread t0( &count_zeros, &b[0], a, 0,    mid1 );
  std::thread t1( &count_zeros, &b[1], a, mid1, mid2 );
  std::thread t2( &count_zeros, &b[2], a, mid2, mid3 );
  count_zeros( &b[3], a, mid3, size );

  // Wait for all threads to finish
  t0.join();
  t1.join();
  t2.join();

  // Serial reduction
  int c = 0;
  for ( int i = 0; i < 4; i++ )
    c += b[i];

  return 0;
}
```

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?

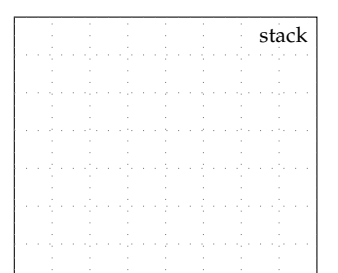
2. C++ Atomics

- What if two threads increment the same variable?




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

```
#include <atomic>

template <>
class atomic<int>
{
 public:
  // constructors
  atomic( int );

  // overloaded operators
  int operator++(int);
  int operator++();
  int operator--(int);
  int operator--();
  int operator+=( int v );
  int operator-=( int v );
  int operator&=( int v );
  int operator|=( int v );
  int operator^=( int v );

  // atomic operations
  int fetch_add( int v );
  int fetch_sub( int v );
  int fetch_and( int v );
  int fetch_or ( int v );
  int fetch_xor( int v );
  ...

 private:
  int m_data;
}
```

```
#include <atomic>

template <>
atomic<int>::atomic( int v )
{
  m_data = v;
}

// pseudo-code, must use special
// hardware instructions to
// guarantee all member functions
// are executed atomically!

template <>
int atomic<int>::operator++(int)
{
  int prev = m_data;
  m_data = m_data + 1;
  return prev;
}

template <>
int atomic<int>::fetch_or( int v )
{
  int prev = m_data;
  m_data = m_data | v;
  return prev;
}
```

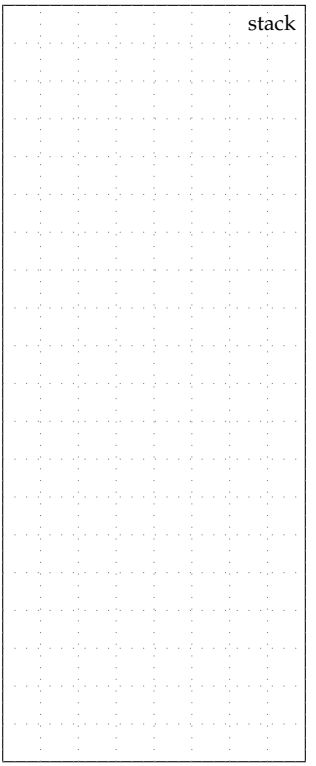
```cpp
#include <thread>
#include <atomic>

void incr( std::atomic<int>* x_p )
{
  (*x_p)++; // guaranteed to execute atomically
}

int main( void )
{
  std::atomic<int> a(0);
  std::thread t( &incr, &a );

  incr( &a );

  t.join();
  return 0;
}
```

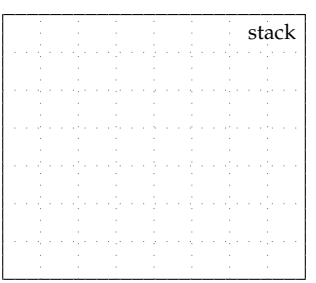
https://godbolt.org/g/bBeRzh

- 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 aquire lock
  - 2. once aquired, execute critical section
  - 3. thread releases lock

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





Encapsulate lock into a mutex

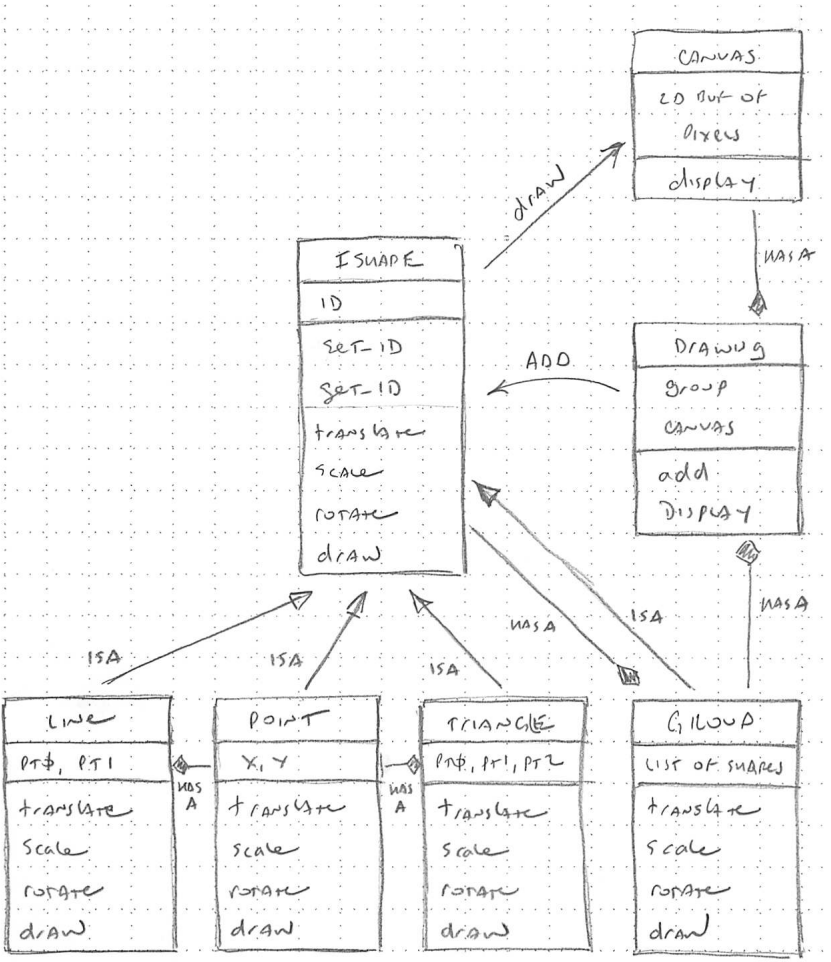
```
1  class Mutex
2  {
3   public:
4    Mutex() { m_lock = 0; }
5
6    void lock()
7    {
8      while ( m_lock.fetch_or(1) == 1 ) { }
9    }
10
11   void unlock() { m_lock = 0; }
12  private:
13   std::atomic<int> m_lock;
14 };
15
16 void incr( int* x_p, Mutex* m_p )
17 {
18   m_p->lock();
19
20   *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

```
class Group : public IShape
{
 public:
  ...

  void draw( Canvas* canvas ) const
  {
    assert( canvas != NULL );

    // Use serial version if fewer than 1000 shapes

    if ( m_shapes_size < 1000 ) {
      for ( int i = 0; i < m_shapes_size; i++ )
        m_shapes[i]->draw( canvas );
    }

    // Use parallel version if 1000 or more shapes

    else {
      int middle = m_shapes_size/2;

      // Child thread draws first half of shapes
      std::thread t( [&](){
        for ( int i = 0; i < middle; i++ )
          m_shapes[i]->draw( canvas );
      });

      // Parent thread draws second half of shapes
      for ( int i = middle; i < m_shapes_size; i++ )
        m_shapes[i]->draw( canvas );

      t.join();
    }
  }
}
```