ECE 2400 Computer Systems Programming Spring 2025

Topic 16: Concurrent Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2025-04-30-11-54

zyBooks logo indicates readings and coding labs 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

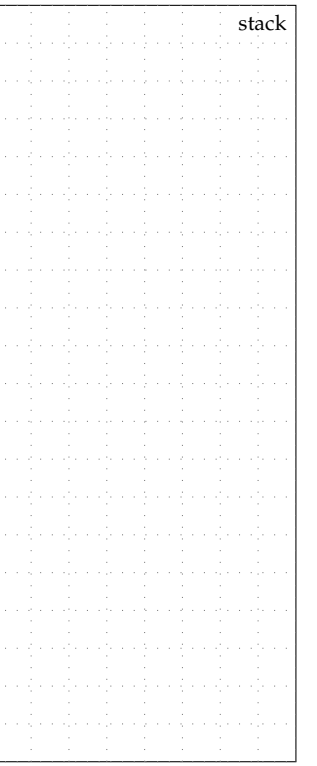
Database Transactions

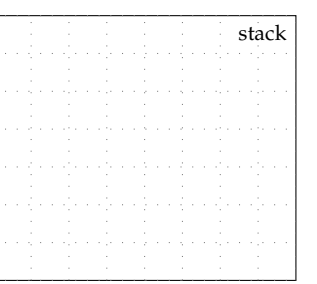
1. C++ Threads

- 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 in the C++ standard lib (include `<thread>` header)
  - `std::thread` objects created using function pointers, functors, or lambdas
  - The pointer to the thread's stack will be its primary member field

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

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





- Use C++ functor to create a thread

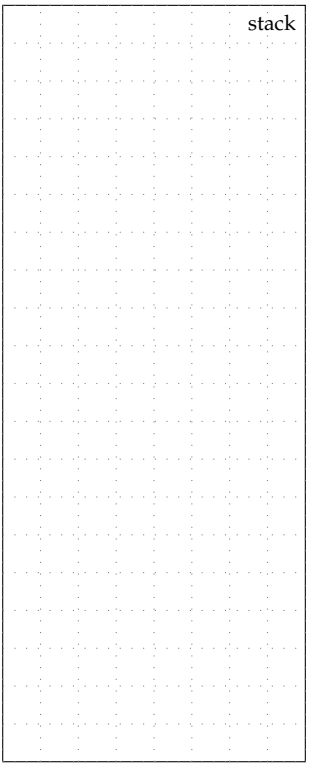
```
class Incr
{
 public:
  void operator()( int* x_p )
  {
    int y = *x_p;
    int z = y + 1;
    *x_p = z;
  }
};

int main( void )
{
  int a = 0;

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

  t.join();

  return 0;
}
```

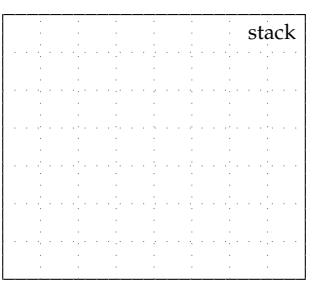
- Use C++ lambda to create a thread

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

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

  t.join();

  return 0;
}
```

- Limitations on the function object used to create a thread
  - Must have a `void` return type
  - Avoid member function pointers
  - Avoid call-by-reference parameters (use call-by-value or call-by-pointer)

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





Parallel Vector-Vector Add

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

Parallel Count Zeros

```
void count_nonzeros( int* x_p, int* y, int first, int last )
{
  int count = 0;
  for ( int i = first; i < last; i++ )
    if ( y[i] != 0 )
      count++;
  *x_p = count;
}

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

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

  // Privatize: Array stores results from each thread
  int b[] = { 0, 0, 0, 0 };

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

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

  // Reduce: Main thread accumulates results from each thread
  int c = 0;
  for ( int i = 0; i < 4; i++ )
    c += b[i];

  return 0;
}
```

Complexity Analysis

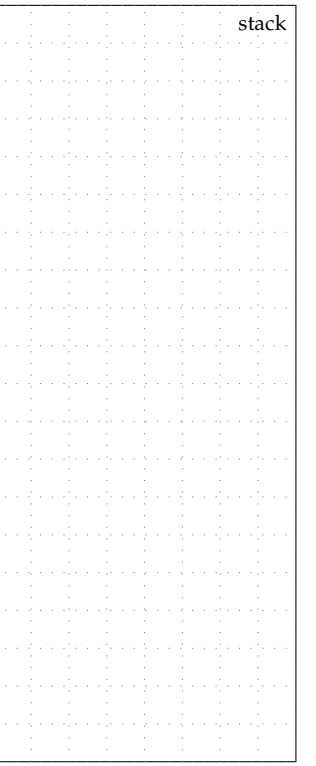
What is the execution time and time complexity as a function of N (size of array) with P (number of processors) as a key constant parameter?

2. C++ Atomics

- Previous approach has each thread write result into private copy, then results are reduced serially or in parallel (privatize-and-reduce)
- What if each thread increments a single shared count?

```
1  void count_nonzeros( int* x_p, int* y, int first, int last )
2  {
3    for ( int i = first; i < last; i++ ) {
4      if ( y[i] != 0 ) {
5        int y = *x_p;
6        int z = y + 1;
7        *x_p  = z;
8      }
9    }
10 }
11
12 int main( void )
13 {
14   const int n = ...
15   int a[n] = { ... };
16
17   int mid1 = 1*(n/4);
18   int mid2 = 2*(n/4);
19   int mid3 = 3*(n/4);
20
21   // Single variable to store shared count
22   int b = 0;
23
24   // Count zeros in each partition in parallel
25   std::thread t0( &count_nonzeros, &b, a, 0,    mid1 );
26   std::thread t1( &count_nonzeros, &b, a, mid1, mid2 );
27   std::thread t2( &count_nonzeros, &b, a, mid2, mid3 );
28   count_nonzeros( &b, a, mid3, size );
29
30   // Wait for all threads to finish
31   t0.join();
32   t1.join();
33   t2.join();
34
35   return 0;
36 }
```

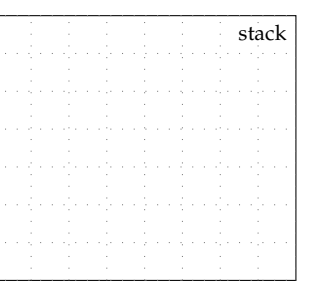
```
    main
t   thread
01  void incr( int* x_p )
02  {
03    int y = *x_p;
04    int z = y + 1;
05    *x_p  = z;
06  }
07
08  int main( void )
09  {
10    int a = 0;
11    std::thread t( &incr, &a );
12
13    incr( &a );
14
15    t.join();
16    return 0;
17  }
```



- Is a single C++ statement atomic?

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

https://godbolt.org/g/zXLFXE



- Use object-oriented & generic programming to implement atomics
  - `std::atomic` is a class in the C++ standard lib (include `<atomic>` header)
  - `std::atomic` objects are thin wrappers around a selected primitive types which restrict access to a few key operators and member functions
  - Uses special hardware instructions to ensure atomicity

```
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;
}
```

```
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;
}
```

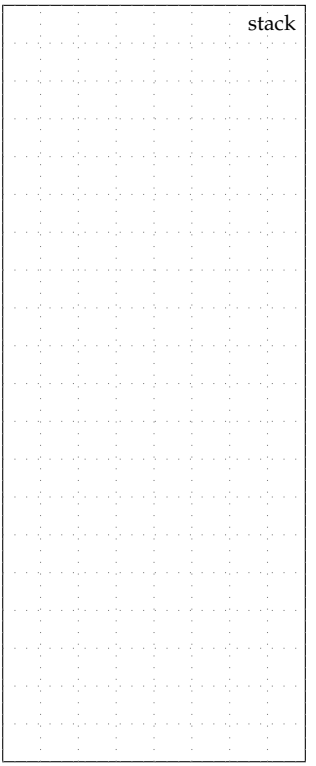
```
1  void incr( std::atomic<int>* x_p )
2  {
3    (*x_p)++; // guaranteed to execute atomically
4  }
5
6  int main( void )
7  {
8    std::atomic<int> a(0);
9    std::thread t( &incr, &a );
10
11   incr( &a );
12
13   t.join();
14   return 0;
15 }
```

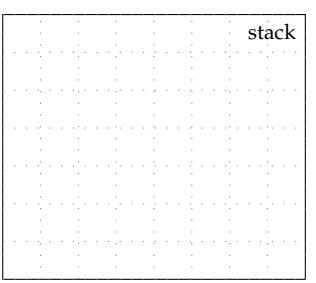
https://godbolt.org/g/bBeRzh

```
1  void count_nonzeros( std::atomic<int>* x_p,
2                       int* y, int first, int last )
3  {
4    for ( int i = first; i < last; i++ ) {
5      if ( y[i] != 0 )
6        (*x_p)++
7    }
8  }
```

- 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

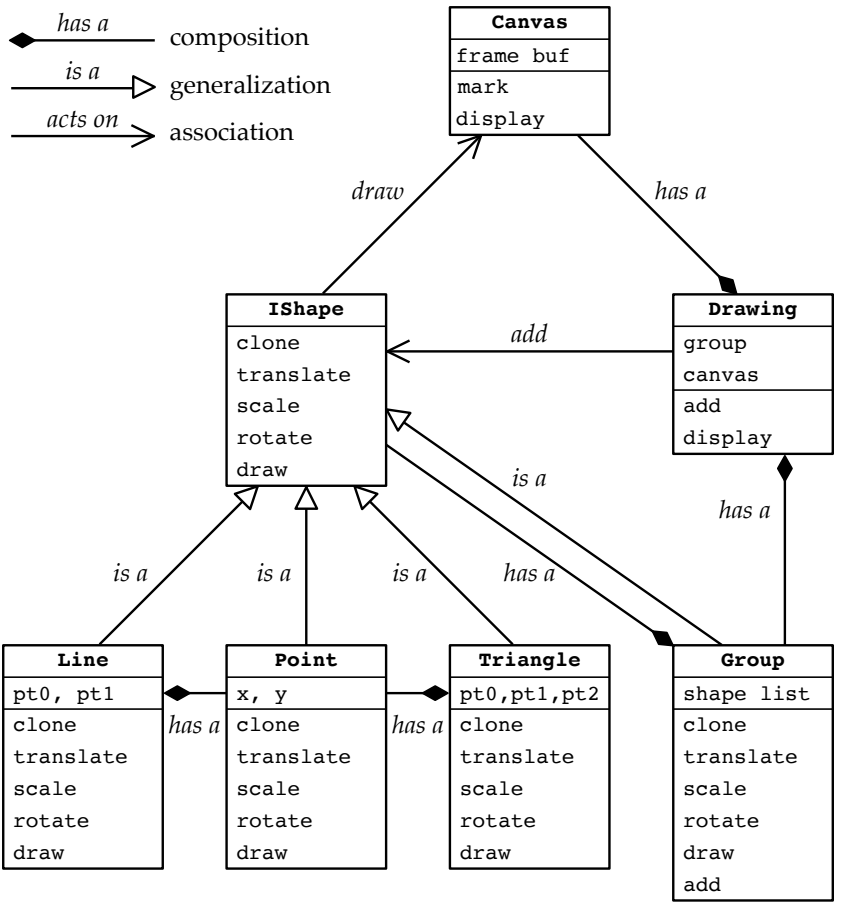
```
1  class Mutex
2  {
3   public:
4    Mutex() { m_lock = 0; }
5
6    void lock()
7    {
8      while ( m_lock.fetch_or(1) == 1 ) { }
9    }
10
11   void unlock() { m_lock = 0; }
12  private:
13   std::atomic<int> m_lock;
14 };
15
16 void incr( int* x_p, Mutex* m_p )
17 {
18   m_p->lock();
19
20  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

Manage mutex as scope-bound resource

- What if we forget to unlock mutex? What if there is an exception?
- 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 mid = m_shapes_size/2;

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

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

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