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)

<table>
<thead>
<tr>
<th>Processing an Array</th>
<th>Sorting an Array</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graphical User Interface</th>
<th>Database Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

```cpp
#include <thread>

class thread
{
public:

  template <typename Func, typename Arg0>
  thread(Func f, Arg0 a0)
  {
    // create new stack
    // set sp to point to new stack
    // start executing function f(a0) using new stack
    // return without waiting for function f to finish
  }

  void join()
  {
    // return when function f is finished
  }

private:
  stack_ptr_t sp;
};
```
1. C++ Threads

```cpp
#include <thread>

void incr( int* x_p )
{
    int y = *x_p;
    int z = y + 1;
    *x_p = z;
}

int main( void )
{
    int a = 0;
    std::thread t( &incr, &a );
    t.join();
    return 0;
}
```
• Use C++ functor to create a thread

```cpp
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;
  std::thread t( &incr );
  t.join();
  return 0;
}
```

• Use C++ lambda to create a thread

```cpp
int main( void )
{
  int a = 0;
  std::thread t( [&]() {
    int y = a;
    int z = y + 1;
    a = z;
  });
  t.join();
  return 0;
}
```
```cpp
#include <thread>

void avg(int* z_p, int x, int y)
{
    int sum = x + y;
    *z_p = sum / 2;
}

int main(void)
{
    int a;
    std::thread t(&avg, &a, 5, 10);
    int b;
    avg(&b, 10, 15);
    t.join();
    return 0;
}
```
#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;
}
1. C++ Threads

Parallel Count Zeros

```cpp
#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?

```cpp
#include <thread>

void incr( int* x_p )
{
    int y = *x_p;
    int z = y + 1;
    *x_p = z;
}

int main( void )
{
    int a = 0;
    std::thread t( &incr, &a );
    incr( &a );
    t.join();
    return 0;
}
```

• Is a single C++ statement atomic?

```cpp
void incr( int* x_p )
{
    (*x_p)++;
}
```

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.

```cpp
#include <atomic>

template <>
class atomic<int> {
public:
    // constructors
    atomic( int v ) {
        m_data = v;
    }

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

    // 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 <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 acquire lock
  – 2. once acquired, execute critical section
  – 3. thread releases lock

```cpp
void incr( int* x_p,
           std::atomic<int>* y_p )
{
  // acquire lock
  while ( y_p->fetch_or(1) == 1 )
  {
  }
  *x_p = foo(*x_p);
  // release lock
  *y_p = 0;
}

int main( void )
{
  int a = 0;
  std::atomic<int> b(0);
  std::thread t( &incr, &a, &b );
  incr( &a, &b );
  t.join();
  return 0;
}
```
Encapsulate lock into a mutex

```cpp
class Mutex {
public:
    Mutex() { m_lock = 0; }

    void lock() {
        while ( m_lock.fetch_or(1) == 1 ) { }
    }

    void unlock() { m_lock = 0; }

private:
    std::atomic<int> m_lock;
};

void incr( int* x_p, Mutex* m_p )
{
    m_p->lock();
    *x_p = foo(*x_p);
    m_p->unlock();
}

int main( void )
{
    int a = 0;
    Mutex m;
    std::thread t( &incr, &a, &m );
    incr( &a, &m );
    t.join();
    return 0;
}
```
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

```cpp
class LockGuard
{
  public:
  
  LockGuard( Mutex* m )
  {
    m_mutex_p = m;
    m_mutex_p->lock();
  }

  ~LockGuard()
  {
    m_mutex_p->unlock();
  }

  private:
    Mutex* m_mutex_p;
};

void incr( int* x_p, Mutex* m_p )
{
  LockGuard guard(m_p);
  *x_p = foo(*x_p);
}
```
3. Drawing Framework Case Study
• Use concurrent programming to accelerate drawing many shapes

```cpp
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();
    }
}
```