ECE 2400 Computer Systems Programming
Fall 2018

Topic 15: Concurrent Programming

School of Electrical and Computer Engineering
Cornell University

revision: 2018-11-26-09-05

1 C++ Threads 3

2 C++ Atomics 10
- 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>
<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>
<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;

};
```
```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;
}
```
1. C++ Threads

- 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;

  Incr incr(&a);
  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;
}
```
1. C++ Threads

```cpp
#include <thread>

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

int main()
{
    int a;
    std::thread t(avg, &a, 5, 10);
    int b;
    avg(&b, 10, 15);
    t.join();
    return 0;
}
```
Parallel Vector-Vector Add

```cpp
#include <thread>

void vvadd( int dest[], int src0[], int src1[],
            size_t lo, size_t hi )
{
    for ( size_t 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];

    size_t middle = size/2;
    std::thread t( vvadd, dest, src0, src1, 0, middle );
    vvadd( dest, src0, src1, middle, size );
    t.join();
    return 0;
}
```
#include <thread>

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

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

    // Sort each partition in parallel

    std::thread t0( mergesort, arr, 0, mid1 );
    std::thread t1( mergesort, arr, mid1, mid2 );
    std::thread t2( mergesort, arr, mid2, mid3 );

    mergesort( arr, mid3, mid4 );

    t0.join();
    t1.join();
    t2.join();

    // Serial merge

    merge( arr, 0, mid1, mid2 ); // merge first two partitions
    merge( arr, mid2, mid3, size ); // merge second two partitions
    merge( arr, 0, mid2, size ); // merge first and second half
    return 0;
}

Parallel Merge Sort

...
Complexity Analysis

What is the execution time complexity as a function of N (size of array) and P (number of processors) using big-O notation?
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
2. C++ Atomics

Using atomic operations

```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

std::atomic<T> member functions

```cpp
operator++
operator--
operator+=
operator-=
operator&=
operator|=
operator^=
```

• What if we want to do something more complicated than this?
• How can we ensure in general a 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) {}
    // release lock
    *y_p = 0;
    *x_p = foo(*x_p);
}

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
#include <atomic>

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 ties resources to object lifetime

```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);
}
```