C++11 Threads

olsen
software ltd

Contents

1. Key concepts
2. Thread safety
3. Additional techniques

1. Key Concepts

- Overview of multithreading in C++11
- Using lambda expressions
- Working with the current thread

Overview of Multithreading in C++11

- C++11 introduces a new thread library
  - Allows you to start and manage threads
  - Provides synchronization utilities such as locks and mutexes

- To create a new thread in C++11:

```
#include <thread>
#include <iostream>

void myFunc()
{
    std::cout << “Hello from another thread!" << std::endl;
}

int main()
{
    std::thread t1(myFunc);
    …
    t1.join();

    return 0;
}
```

Using Lambda Expressions

- The `std::thread` constructor allows you to use a lambda expression to specify the code to run in the thread
  - A lambda expression is like an anonymous inline function, as we saw earlier in the course

- Example:

```
#include <thread>
#include <iostream>

int main()
{
    std::thread t1([](){
        std::cout << "Hello from another thread!" << std::endl;
    });
    …
    t1.join();

    return 0;
}
```

Working with the Current Thread

- The `std::this_thread` namespace has various functions that enable you to work with the current thread
  - `get_id()` gets the id of current thread:

```
std::thread::id id = std::this_thread::get_id();
```

  - `yield()` yields control in current thread:

```
std::this_thread::yield();
```

  - `sleep_for()` blocks current thread for a specified duration:

```
std::this_thread::sleep_for(sleep_duration);
```

  - `sleep_until()` blocks current thread until the specified time:

```
std::this_thread::sleep_until(sleep_time);
```

2. Thread Safety

- Protecting data from simultaneous access
- Example of using a mutex
- Additional types of mutex
- Exception-safe lock management
- General lock management
- Specifying a locking strategy
- Acquiring multiple locks simultaneously

Protecting Data from Simultaneous Access

- It is essential to protect shared data from being simultaneously accessed by multiple threads
- C++ provides the `std::mutex` class for this purpose
  - Defined in the `<mutex>` header file
  - You create a `std::mutex` object to protect some piece of data
- When a thread wants to access the data:
  - Call `lock()` or `try_lock()` to acquire the mutex
  - This call blocks if another thread has already acquired the mutex
- When a thread has finished using the data:
  - Call `unlock()` to release the mutex
  - Allows another blocked thread to acquire the mutex and unblock

Example of Using a Mutex

```
#include <mutex>

class BankAccount
{
    std::mutex mutex;
    double balance;

public:
    BankAccount() : balance(0) {}

    void Deposit(double amount)
    {
        mutex.lock();
        balance += amount;
        mutex.unlock();
    }

    void Withdraw(double amount)
    {
        mutex.lock();
        balance -= amount;
        mutex.unlock();
    }
};
```

Additional Types of Mutex

- There are several variations on the basic `std::mutex`…
- `std::timed_mutex`
  - Enables you to specify how long to wait to acquire the mutex
  - Via `try_lock_for()` and `try_lock_until()`
- `std::recursive_mutex`
  - Keeps count of how many times the owning thread has locked it
  - Must be unlocked the same number of times to be released
- `std::recursive_timed_mutex`
  - Combines the capabilities of the two classes above

Exception-Safe Lock Management

- You can use `std::lock_guard` to manage a lock automatically
  - Acquires a mutex on creation
  - Releases a mutex automatically when it goes out of scope

- Example:

```
class BankAccount
{
    …

    void Withdraw(double amount)
    {
        std::lock_guard<std::mutex> guard(mutex);

        if (amount > balance)
        {
            throw std::exception("Insufficient funds");
        }
        balance -= amount;
    }
};
```

General Lock Management

- You can use `std::unique_lock` to help manage a lock
  - Takes a mutex object in the constructor
  - Provides wrapper functions for all the common mutex operations

```
std::unique_lock<std::mutex> lock(aMutex);
```

- `std::unique_lock` methods for managing state of lock:
  - `swap()`
  - `release()`
  - `operator=()`

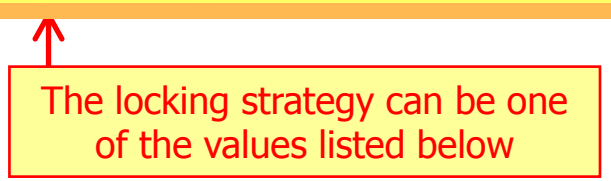
- `std::unique_lock` methods for observing state of lock:
  - `mutex()`
  - `owns_lock()`
  - `operator bool()`

Specifying a Locking Strategy

- When you create a std::lock_guard or std::unique_lock, you can specify a locking a strategy parameter

```
std::lock_guard<std::mutex> guard(aMutex, locking_strategy);
```

```
std::unique_lock<std::mutex> lock(aMutex, locking_strategy);
```



- `std::adopt_lock`
  - Assume the calling thread already owns the lock
- `std::try_to_lock`
  - Try to acquire ownership of the mutex without blocking
- `std::defer_lock`
  - Do not acquire ownership of the mutex

Acquiring Multiple Locks Simultaneously

- Some programming scenarios require you to acquire several locks at the same time
  - E.g. acquire locks on several objects so you can update all of them

```
std::mutex m1, m2;  // Etc…

std::lock(m1, m2);
…
m1.unlock();
m2.unlock();
```

- `std::lock()`

```
std::mutex m1, m2;  // Etc…

int r = std::try_lock(m1, m2);
if (r == -1)
{
  …
  m1.unlock();
  m2.unlock();
}
```

- `std::try_lock()`
  - Returns -1 if all locks acquired
  - Otherwise, releases all locks that were acquired, and returns the index of the first lock not acquired

3. Additional Techniques

- Atomic variables
- Coordinating threads with condition variables
- How to use condition variables
- Example of condition variables
- Calling functions asynchronously

Atomic Variables

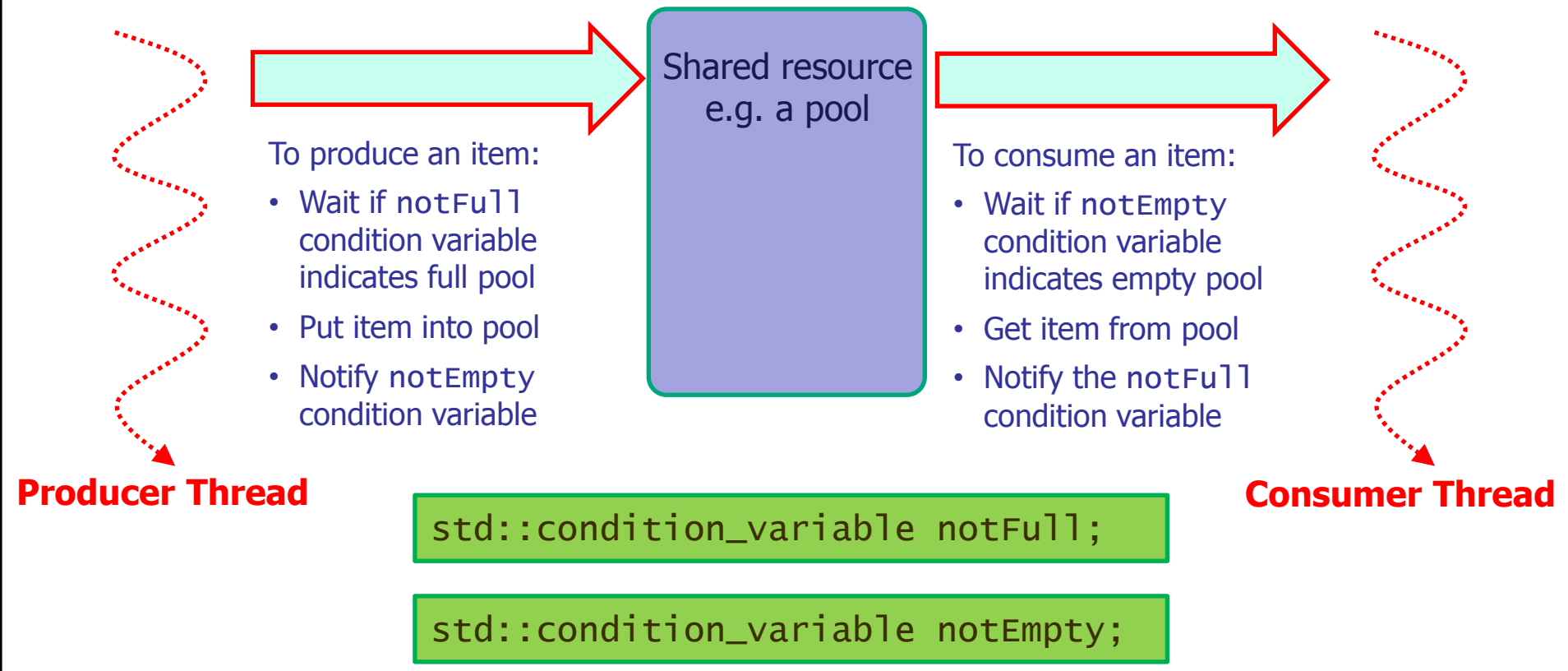
- You can use `std::atomic` to wrap an object as an atomic type
  - Ensures all operations on the object are atomic, i.e. thread-safe

```
#include <atomic>

std::atomic<int> value;

++value;
--value;
```

- `std::atomic` include various helper methods, including:
  - `store()`, `load()`, `exchange()`
  - `fetch_add()`, `fetch_sub()`
  - `fetch_and()`, `fetch_or()`, `fetch_xor()`
  - `operator++`, `operator--`, `operator+=`, `operator-=`
  - `operator&=`, `operator|=`, `operator^=`

Coordinating Threads with Condition Variables

- You can use a `std::condition_variable` to coordinate threads in a collaborative manner
  - E.g. to implement the producer/consumer pattern




```
std::condition_variable notFull;
```

```
std::condition_variable notEmpty;
```

How to use Condition Variables

- Before you can use a condition variable:
  - Create a `std::unique_lock` object to acquire (i.e. lock) a mutex
- When you've acquired the mutex, call a wait function on a `std::condition_variable` to see if it's OK to proceed
  - Param 1: the `std::unique_lock` object
  - Param 2: optional predicate, returns `false` to keep waiting
- When the waiting is over:
  - Do whatever work you want to do in your thread
- When you've finished doing your work, call a "notify" function on a `std::condition_variable`
  - Signals other (waiting) threads that they might be able to proceed

Example of Condition Variables

General variables:

```
int count, size;
std::mutex mutex;
std::condition_variable notFull, notEmpty;
```

Producer thread code:

```
std::unique_lock<std::mutex> lock(mutex);
notFull.wait(lock, [](){ return count != size; });

… insert item …

notEmpty.notify_one();
```

Consumer thread code:

```
std::unique_lock<std::mutex> lock(mutex);
notEmpty.wait(lock, [](){ return count != 0; });

… retrieve item …

notFull.notify_one();
```

Calling Functions Asynchronously

- You can use the `std::async()` template function to call a function asynchronously

- To use `std::async()`:
  - Specify the function you want to invoke, plus a list of arguments
  - Optionally, you can also specify a launch strategy
  - Returns a `std::future` that will eventually hold function result

```
std::future std::async(function, argument_list)
```

```
std::future std::async(std::launch, function, argument_list)
```

- `std::launch` is a bitwise enum with these values:
  - `std::launch::async`
  - `std::launch::deferred`

Any Questions?