Lecture 3: C++ Thread Programming

Tsung-Wei (TW) Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

CPU Parallelism using C++ Thread

- Creating a std::thread
- Caring about const- and reference-semantics (std::cref and std::ref)
- Solving race conditions (std::mutex)
- Getting return values (std::future<T>)
- Allowing threads to complete / cleanup (std::thread::join or std::thread::detach)
- General Algorithm: partitioning data
- Style/Cleanup – Using C++ Lambdas

Creating a std::thread

```
#include <thread>
using std::thread;

thread (fun, args...)
```

- **fun -** a function we wish to have a thread run
- **args... -** any number of arguments we wish to pass to fun

https://en.cppreference.com/w/cpp/thread/thread/thread

```
thread t1(printf, "Hello from other thread", 1);
thread t2(vector<int>::push_back, v, 1);        // calls v.push_back(1)
```

Caring about const- and Reference-Semantics

- Functions can have varying signature types
- It can be very difficult to disambiguate between function overloads
- Threads can ***only*** be constructed with value-semantics
- Idea: Introduce a *reference wrapper* type which encapsulates references and const-references
  - https://en.cppreference.com/w/cpp/utility/functional/ref
  - `#include <functional>`
  - std::cref for const-reference
  - std::ref for reference

std::cref and std::ref

```
#include <functional>
using std::ref;
using std::cref;
void myFunction(const vector<int>& v, int& result);
...
// Usage:
thread t(myFunction, cref(vec), ref(ans));
```

Not Using Reference Wrappers…

```
/usr/include/c++/9.2.0/thread: In instantiation of ‘std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void
(&)(int&); _Args = {int&}; <template-parameter-1-3> = void]’:
example.cpp:13:20:   required from here
/usr/include/c++/9.2.0/thread:120:44: error: static assertion failed: std::thread arguments must be invocable after conversion
to rvalues
  120 |           typename decay<_Args>::type...>::value,
      |                                            ^~~~~
/usr/include/c++/9.2.0/thread: In instantiation of ‘struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >’:
/usr/include/c++/9.2.0/thread:131:22:   required from ‘std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void
(&)(int&); _Args = {int&}; <template-parameter-1-3> = void]’
example.cpp:13:20:   required from here
/usr/include/c++/9.2.0/thread:243:4: error: no type named ‘type’ in ‘struct std::thread::_Invoker<std::tuple<void (*)(int&),
int> >::__result<std::tuple<void (*)(int&), int> >’
  243 |    _M_invoke(_Index_tuple<_Ind...>)
      |    ^~~~~~~~~
/usr/include/c++/9.2.0/thread:247:2: error: no type named ‘type’ in ‘struct std::thread::_Invoker<std::tuple<void (*)(int&),
int> >::__result<std::tuple<void (*)(int&), int> >’
  247 |  operator()()
      |  ^~~~~~~~
make: *** [<builtin>: example] Error 1
```

Not Using Reference Wrappers…

```
/usr/include/c++/9.2.0/thread: In instantiation of ‘std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void
(&)(int&); _Args = {int&}; <template-parameter-1-3> = void]’:
example.cpp:13:20:   required from here
/usr/include/c++/9.2.0/thread:120:44: error: static assertion failed: std::thread arguments must be invocable after conversion
to rvalues
  120 |           typename decay<_Args>::type...>::value,
      |                                            ^~~~~
/usr/include/c++/9.2.0/thread: In instantiation of ‘struct std::thread::_Invoker<std::tuple<void (*)(int&), int> >’:
/usr/include/c++/9.2.0/thread:131:22:   required from ‘std::thread::thread(_Callable&&, _Args&& ...) [with _Callable = void
(&)(int&); _Args = {int&}; <template-parameter-1-3> = void]’
example.cpp:13:20:   required from here
/usr/include/c++/9.2.0/thread:243:4: error: no type named ‘type’ in ‘struct std::thread::_Invoker<std::tuple<void (*)(int&),
int> >::__result<std::tuple<void (*)(int&), int> >’
  243 |    _M_invoke(_Index_tuple<_Ind...>)
      |    ^~~~~~~~~
/usr/include/c++/9.2.0/thread:247:2: error: no type named ‘type’ in ‘struct std::thread::_Invoker<std::tuple<void (*)(int&),
int> >::__result<std::tuple<void (*)(int&), int> >’
  247 |  operator()()
      |  ^~~~~~~~
make: *** [<builtin>: example] Error 1
```

Solving Race Conditions

- Consider the following:

```
thread t1(std::vector<int>::push_back, ref(v), 1);
thread t2(std::vector<int>::push_back, ref(v), 2);
v.push_back(3);
t1.join();
t2.join();
```

- ”v” can be modified **concurrently** by t1, t2, or the “main” thread!
- push_back() is ***NOT*** thread-safe!

Solving Race Conditions

- Idea: only allow **<u>one thread</u>** in a region at a time
  - Term: mutual exclusion
- In programming, we use a special object to represent this idea – mutex
  - **Mutex** = **<u>mut</u>**ual **<u>ex</u>**clusion
- Mutexes can “lock” and “unlock”

```
#include <mutex>
using std::mutex;
```

std::mutex

```
// m is shared among all threads
mutex m;
// This is a place where multiple threads can be

m.lock(); // begin mutex region
// only ONE thread is allowed in here at a time
m.unlock(); // end of mutex region
```

Making a std::mutex shared

- Option 1: pass as a parameter to a “thread-safe function”

```
template <typename T>
Void safe_push_back(vector<T>& v, const T& value, mutex& lock)
{
  lock.lock();
  v.push_back(value);
  lock.unlock();
}
```

- Option 2: make it global
  - LOL JK **DON’T EVER DO THIS**

Making a std::mutex shared

- Calling the safe function:

```
vector<int> v = …;
mutex m;

thread t1(safe_push_back<int>, ref(v), 1, ref(m));
thread t2(safe_push_back<int>, ref(v), 2, ref(m));
safe_push_back(v, 3, m);
t1.join();
t2.join();
```

Getting Return Values

- Up until this point, threads could call functions but not “return” anything
- We have functions that should return a value!
- Idea: threads run and take some time – we will get its result in the **future**
- Solution: introduce the concept of a future

```
#include <future>
using std::future;
```

Creating a std::future

- There are a few ways in C++ to create a future, but we will only focus on ONE
  - std::thread allowed us to create a thread which we know would run asynchronously to our main thread
  - We want to asynchronously run an get a **future**
  - Introducing: std::async

```
using std::async;
// similar to thread
async (std::launch::async, Fun, Args...)
```

std::future – “get()”-ing the result

```
int rand_between (int low, int high) {
  static std::minstd_rand rng{0};
  return low + rng () % (high - low);
}

future<int> result = async (
  std::launch::async, rand_between, 0, 10);
cout << result.get() << endl;
```

Managing ALL THE THREADS

Usually we will have some procedure that does the following:

1. Create a bunch of threads
2. Assign them to do a piece of the work
3. Wait for them to finish
4. ???
5. Profit

Managing ALL THE THREADS – CREATION

- Creating a bunch of threads
  - And have them do a work

```
vector<thread> threads;
for (int tid = 0; tid < numThreads; ++tid) {
  threads.emplace_back ( /* args to thread ctor */ );
}
```

Managing ALL THE FUTURES – Wait/Cleanup

- Wait for them to finish

```
for (auto& t : threads) {
  t.join();
}
```

Managing FUTURES – CREATION

- Creating a bunch of asyncs
  - And have them do a work

```
vector<future<results>> threads;
for (int tid = 0; tid < numThreads; ++tid) {
  threads.emplace_back (std::launch::deferred,
                        /* args to thread ctor */ );
}
```

Managing FUTURES – Wait/Cleanup

- Wait for them to finish

```
for (auto& t : threads) {
  auto result = t.get();
  // do something with result
}
```

Partitioning Data

Given a range of values from [low, high)

- We would like to be able to divide the work
- Work should be <u>block-distributed</u>
- Work should be <u>evenly-divided</u>

- **<u>Goal:</u> Do this efficiently**
- **<u>Strategy:</u> Consider "size", number of "workers", and current "worker ID"**

Partitioning Data

- A “range” is defined as a lower-bound and upper-bound
  - Initially, this is often [low, high) or [0, N)
- When we have <u>two</u> workers, we want to partition as such:
  - Worker 0: [0, N/2)
  - Worker 1: [N/2, N)
- When we have <u>three</u> workers…
  - Worker 0: [0, N/3)
  - Worker 1: [N/3, 2*N/3)
  - Worker 2: [2*N/3, N)

Partitioning Data

```
int get_index (int elems, int id, int total)
{
  // ideally: elems * (id / total)
  return static_cast<long long>(elems) * id / total;
}
```

Partitioning Data

- Our ranges can be generalized… for N elements with P workers:
  - **Worker 0:**
    - Low: get_index(N, 0, P)
    - High: get_index(N, 1, P)
  - **Worker 1:**
    - Low: get_index(N, 1, P)
    - High: get_index(N, 2, P)
  - **Worker P-1:**
    - Low: get_index(N, P - 1, P)
    - High: get_index(N, P, P)

Worker i:

- Low: get_index(N, i, P)
- High: get_index(N, i + 1, P)

Using C++ Lambdas

- Consider the following scenario:
  - You have a vector of ints you'd like to populate
  - You want to distribute the work across threads
  - You want each thread to add K copies of itself to the vector

Definition:

```
void populate (vector<int>& v, mutex& m, int K, int tid);
// v, m, and K are the same for ALL
// have to use ref() for v and m
```

Usage:

```
thread t1 (populate, ref(vec), ref(mut), K, tid);
```

Using C++ Lambdas

```
// vec, mut, and K are actual parameters
auto populate = [&vec, &mut, K] (int tid) {
  for (int i = 0; i < K; ++i) {
    mut.lock();
    vec.push_back(tid);
    mut.unlock();
  }
};
```

Using C++ Lambdas

```
vector<int> vec;
mutex mut;
int K = …;
auto populate = … /* lambda definition */;
for (int tid = 0; tid < P; ++tid) {
  threads.emplace_back (populate, tid);
}
```

Using C++ Lambdas

- Before

```
void populate (vector<int>& v, mutex& m, int K, int tid);
threads.emplace_back(populate, ref(vec), ref(mut), K, tid);
```

- After

```
auto pop = [&vec, &mut, K] (int tid) {
  return populate (vec, mut, K, tid);
};
threads.emplace_back(pop, tid);
```