Making code thread-safe

Kyle J. Knoepfel
25 June 2019
LArSoft Workshop 2019
So you’re going to make your code thread-safe…
So you’re going to make your code thread-safe…

- The difficulty of this task depends on the context
So you’re going to make your code thread-safe…

- The difficulty of this task **depends on the context**

- What language are you using?
  - Multi-threading in (e.g.) C++ is harder
  - Multi-threading in (e.g.) Go, Rust, Haskell is easier

- Are you starting from scratch or retrofitting code?

- Does it make sense for the code in question to be multi-threaded?
So you’re going to make your code thread-safe…

• The difficulty of this task depends on the context

• What language are you using?
  – Multi-threading in (e.g.) C++ is harder
  – Multi-threading in (e.g.) Go, Rust, Haskell is easier

• Are you starting from scratch or retrofitting code?

• Does it make sense for the code in question to be multi-threaded?

When writing multi-threaded code, you should always ask:

What’s the context in which this function will be called?
Thread-safety matrix
# Thread-safety matrix

<table>
<thead>
<tr>
<th>Is the object shared across threads?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
</tbody>
</table>

- **Yes**
- **No**
Thread-safety matrix

<table>
<thead>
<tr>
<th>Is the object shared across threads?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Is the object mutable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
<tr>
<td>Is the object shared across threads?</td>
</tr>
<tr>
<td>------------------------------------</td>
</tr>
<tr>
<td>Is the object mutable?</td>
</tr>
<tr>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
</tr>
</tbody>
</table>
## Thread-safety matrix

<table>
<thead>
<tr>
<th>Is the object shared across threads?</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is the object mutable?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>OK</td>
</tr>
<tr>
<td>Synchronous access</td>
<td></td>
<td>Data races guaranteed</td>
</tr>
<tr>
<td>Asynchronous access</td>
<td></td>
<td>OK</td>
</tr>
</tbody>
</table>

- **OK**: Safe to use in parallel without synchronization
- **Data races guaranteed**: Not safe to use in parallel without synchronization
## Thread-safety matrix

<table>
<thead>
<tr>
<th>Is the object shared across threads?</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>OK</td>
</tr>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Is the object mutable?</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>No</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

- **Synchronous access**
- **Asynchronous access**

**Data races guaranteed**

You must get out of that box!
### Thread-safety matrix

<table>
<thead>
<tr>
<th>Is the object shared across threads?</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is the object mutable?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>Yes</td>
<td>Data races guaranteed</td>
<td>OK</td>
</tr>
</tbody>
</table>

- **Synchronous access**: Sometimes the easiest solution. Requires more memory.
- **Asynchronous access**:
Thread-safety matrix

Is the object shared across threads?

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is the object mutable?

<table>
<thead>
<tr>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Synchronous access

Asynchronous access

Data races guaranteed

Sometimes the easiest solution.
Requires more memory.

Memory savings.
Can be difficult to make things immutable.

OK
# Thread-safety matrix

<table>
<thead>
<tr>
<th>Is the object shared across threads?</th>
<th>Is the object mutable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Synchronous access</strong></td>
<td>Awkward. Will not discuss.</td>
</tr>
<tr>
<td></td>
<td>Data races guaranteed</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Synchronous access</td>
<td>Memory savings. Can be difficult to make things immutable.</td>
</tr>
<tr>
<td></td>
<td>OK</td>
</tr>
</tbody>
</table>

- Sometimes the easiest solution. Requires more memory.
- Sometimes the easiest solution. Requires more memory.
- Is the object shared across threads?
- Is the object mutable?
Thread-safety matrix

Is the object shared across threads?

<table>
<thead>
<tr>
<th></th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Synchronous access</td>
<td>Asynchronous access</td>
</tr>
<tr>
<td></td>
<td>Awkward. Will not discuss.</td>
<td>Data races guaranteed</td>
</tr>
<tr>
<td>No</td>
<td>Sometimes the easiest solution. Requires more memory.</td>
<td>OK</td>
</tr>
<tr>
<td>Is the object mutable?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Memory savings. Can be difficult to make things immutable.</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Making your code thread-safe often requires a combination of methods.
To make your code thread-safe...

- You must know what is shared among threads, and the contexts in which the sharing happens.
To make your code thread-safe...

- **You must know what is shared among threads, and the contexts in which the sharing happens.**

- **Game – Part 1**
  - Thread-safety and free-functions

- **Game – Part 2**
  - Thread-safety and class member functions
Part 1: Is it thread-safe?

The pattern we’ll follow

```c
void test(...) {
    // ...
}

int main() {
    execute_with_10_threads(test, ...);
}
```
void test()
{
}

int main()
{
    execute_with_10_threads(test);
}
Part 1: Is it thread-safe?

```c
void test()
{
}

int main()
{
    execute_with_10_threads(test);
}
```

Yes
Part 1: Is it thread-safe?

```c
void test()
{
    auto i = 42;
}

int main()
{
    execute_with_10_threads(test);
}
```
Part 1: Is it thread-safe?

```cpp
void test()
{
    auto i = 42;
}

int main()
{
    execute_with_10_threads(test);
}
```

Yes

Each thread gets its own stack memory.
Part 1: Is it thread-safe?

```c
void test(int j)
{
    ++j;
}

int main()
{
    auto i = 42;
    execute_with_10_threads(test, i);
}
```
Part 1: Is it thread-safe?

```c
void test(int j)
{
    ++j;
}

int main()
{
    auto i = 42;
    execute_with_10_threads(test, i);
}
```

Yes

The value 42 is copied into j for each thread.
Part 1: Is it thread-safe?

```c
void test()
{
    static int j{0};
    ++j;
}

int main()
{
    execute_with_10_threads(test);
}
```
Part 1: Is it thread-safe?

```c
void test()
{
    static int j{0};
    ++j;
}

int main()
{
    execute_with_10_threads(test);
}
```

No

j is shared across all threads that call test

operator++ requires a read and then write
Part 1: Is it thread-safe?

```c
void test()
{
    static int j{0};
    int k = j;
    ++k;
}

int main()
{
    execute_with_10_threads(test);
}
```
void test()
{
    static int j{0};
    int k = j;
    ++k;
}

int main()
{
    execute_with_10_threads(test);
}
Part 1: Is it thread-safe?

```c
void test() {
    static int j{0};
    int& k = j;
    ++k;
}

int main() {
    execute_with_10_threads(test);
}
```
Part 1: Is it thread-safe?

```c
void test()
{
    static int j{0};
    int& k = j;
    ++k;
}

int main()
{
    execute_with_10_threads(test);
}
```

No

$k$ now refers to a shared object!

`operator++` requires a read and then write
Part 1: Is it thread-safe?

```c
void test()
{
    static int j{0};
    int& k = j;
    ++k;
}

int main()
{
    execute_with_10_threads(test);
}
```

No

k now refers to a shared object!

operator++ requires a read and then write

One character can break a program.
Part 1: Is it thread-safe?

```cpp
void test(std::string const& sentence) {
    auto pos = sentence.find("C++17");
}

int main() {
    std::string sentence{"I love C++17."};
    execute_with_10_threads(test, sentence);
}
```
Part 1: Is it thread-safe?

```cpp
void test(std::string const& sentence)
{
    auto pos = sentence.find("C++17");
}

int main()
{
    std::string sentence{"I love C++17."};
    execute_with_10_threads(test, sentence);
}
```

Yes

In general, calling const-qualified C++ STL member functions is thread-safe... assuming another thread isn’t adjusting the object.
Part 1: Is it thread-safe?

```c
void test()
{
    MyArbitraryType t;
}

int main()
{
    execute_with_10_threads(test);
}
```
Part 1: Is it thread-safe?

```c
void test()
{
    MyArbitraryType t;
}

int main()
{
    execute_with_10_threads(test);
}
```

It depends
Part 1: Is it thread-safe?

```cpp
void test()
{
    MyArbitraryType t;
}

int main()
{
    execute_with_10_threads(test);
}

using MyArbitraryType = int;
```
Part 1: Is it thread-safe?

```c
void test()
{
    MyArbitraryType t;
}

int main()
{
    execute_with_10_threads(test);
}
```

Yes
Part 1: Is it thread-safe?

```c
void test()
{
    MyArbitraryType t;
}

int main()
{
    execute_with_10_threads(test);
}

struct MyArbitraryType {
    MyArbitraryType()
    {
        static int counter;
        ++counter;
    }
};
```
Part 1: Is it thread-safe?

```c
void test() {
    MyArbitraryType t;
}

int main() {
    execute_with_10_threads(test);
}
```

```c
struct MyArbitraryType {
    MyArbitraryType() {
        static int counter;
        ++counter;
    }
};
```

No

Although there is one ‘t’ per thread, the constructor accesses shared memory.
Part 2: Is it thread-safe?

```cpp
class MyClass {
public:
  void test(...) {
    // ...
  }
};

int main() {
  MyClass obj;
  execute_with_10_threads(&MyClass::test, obj, ...);
}
```

The pattern we’ll follow
Part 2: Is it thread-safe?

class MyClass {
public:
    void test()
    {
    }
};

int main()
{
    MyClass obj;
    execute_with_10_threads(&MyClass::test, obj);
}
Part 2: Is it thread-safe?

class MyClass {
public:
    void test()
    {
    }
};

int main()
{
    MyClass obj;
    execute_with_10_threads(&MyClass::test, obj);
}
class MyClass {
    int count_{0};

public:
    void test()
    {
        ++count_;  
    }

};

int main()
{
    MyClass obj;
    execute_with_10_threads(&MyClass::test, obj);
}
Part 2: Is it thread-safe?

```cpp
class MyClass {
    int count_{0};

public:
    void test()
    {
        ++count_;  
    }
};

int main()
{
    MyClass obj;
    execute_with_10_threads(&MyClass::test, obj);
}

No
The class data for obj (count_) is shared across threads.

operator++ requires a read and then write
```
Part 2: Is it thread-safe?

class MyClass {
public:
    void test(MyOtherClass& oc) {
        auto const h = oc.getSomething();
    }
};

int main() {
    MyClass obj;
    MyOtherClass oc;
    execute_with_10_threads(&MyClass::test, obj, oc);
}
Part 2: Is it thread-safe?

```cpp
class MyClass {
    public:
        void test(MyOtherClass& oc) {
            auto const h = oc.getSomething();
        }
};

int main() {
    MyClass obj;
    MyOtherClass oc;
    execute_with_10_threads(&MyClass::test, obj, oc);
}
```

It depends
Part 2: Is it thread-safe?

class MyClass {
    public:
        void test(MyOtherClass& oc) {
            auto const h = oc.getSomething();
        }
    }

int main() {
    MyClass obj;
    MyOtherClass oc;
    execute_with_10_threads(&MyClass::test, obj, oc);
}
Part 2: Is it thread-safe?

class MyClass {
public:
    void test(MyOtherClass& oc) {
        auto const h = oc.getSomething();
    }
};

int main() {
    MyClass obj;
    MyOtherClass oc;
    execute_with_10_threads(&MyClass::test, obj, oc);
}

class MyOtherClass {
public:
    int getSomething() const {
        return 42;
    }
};

Yes
Part 2: Is it thread-safe?

```cpp
class MyClass {
public:
    void test(MyOtherClass& oc)
    {
        auto const h = oc.getSomething();
    }
};

int main()
{
    MyClass obj;
    MyOtherClass oc;
    execute_with_10_threads(&MyClass::test, obj, oc);
}

class MyOtherClass {
public:
    int getSomething() const
    {
        static int i{42};
        ++i;
        return i;
    }
};
```
Part 2: Is it thread-safe?

class MyClass {
public:
  void test(MyOtherClass& oc) {
    auto const h = oc.getSomething();
  }
};

int main() {
  MyClass obj;
  MyOtherClass oc;
  execute_with_10_threads(&MyClass::test, obj, oc);
}

class MyOtherClass {
public:
  int getSomething() const {
    static int i{42};
    ++i;
    return i;
  }
};

No!
Part 2: Is it thread-safe?

class MyClass {
public:
    void test(MyOtherClass& oc)
    {
        auto const h = oc.getSomething();
    }
};

int main()
{
    MyClass obj;
    MyOtherClass oc;
    execute_with_10_threads(&MyClass::test, obj, oc);
}
Part 2: Is it thread-safe?

class MyClass {
    public:
        void produce(art::Event& e)
        {
            auto const h = e.getValidHandle();
        }
};

int main()
{
    MyClass obj;
    art::Event e;
    execute_with_10_threads(&MyClass::produce, obj, e);
}
Determining thread-safety …

• Takes analysis!

• Know what objects are shared.

• Know when they are shared.
Considerations for *art*

- Who owns your module?
Considerations for *art*

- Who owns your module?

- *art* owns the module objects, which are created at run-time based on the configuration you provide.

- You provide the *definition* of the module class:
  - *art* knows very little of your module’s definition
  - *art* calls module functions via C++ polymorphism

- Suppose *art* were to call your *produce* function concurrently on multiple events.
Module example

• We want to create a track from a collection of hits

```cpp
void TrackMaker::produce(art::Event& e) {
    auto const& hits = e.getValidHandle<Hits>(tag_);
    unique_ptr<Track> track = trackFromHits(*hits);
    e.put(move(track));
}
```
Module example

• We want to create a track from a collection of hits

```cpp
void TrackMaker::produce(art::Event& e) {
    auto const& hits = e.getValidHandle<Hits>(tag_);
    unique_ptr<Track> track = trackFromHits(*hits);
    e.put(move(track));
}
```

• Assuming `trackFromHits` does not update any state, then this `produce` function is thread-safe—i.e. it can be called concurrently with different `art::Event` objects.

• Why?
  – `art` guarantees that product retrieval and insertion is thread-safe
  – the produce function above modifies no state of the TrackMaker object
Indications of thread-unsafe C++ code

- Functions (free or member) which access a global object whose state can change, including non-const function-scope static data.
- Functions (free or member) which change the state of objects which were passed as const function arguments (e.g. casting away const on an argument).
- const non-static member functions which modify the state of the object on which they are called (e.g. mutable members, or casting away const on this).
- Pointer member data or data held by member data being passed as a non-const argument to functions.
- const member functions returning values of member variables which are pointers to non-const items.
General guidelines

• Apply `const` liberally
• Avoid using non-`const` static variables in functions/classes
• To the extent possible, do not use the `mutable` keyword
• Use as few global objects as possible
• If you must use a global object, provide only `const`-qualified interface
• Don’t use output arguments
General guidelines

- Apply \texttt{const} liberally
- Avoid using non-\texttt{const static} variables in functions/classes
- To the extent possible, do not use the \texttt{mutable} keyword
- Use as few global objects as possible
- If you must use a global object, provide only \texttt{const}-qualified interface
- Don’t use output arguments

Discouraged

\begin{verbatim}
void fillInts(std::vector<int>& ints) {
    for (size_t i{0}; i < 42; ++i)
        ints.push_back(some_calculation(i));
} std::vector<int> ints;
fillInts(ints);
\end{verbatim}

Encouraged

\begin{verbatim}
auto getInts() {
    std::vector<int> ints;
    for (size_t i{0}; i < 42; ++i)
        ints.push_back(some_calculation(i));
    return ints;
} auto const ints = getInts();
\end{verbatim}
General guidelines

• Apply `const` liberally
• Avoid using non-`const` static variables in functions/classes
• To the extent possible, do not use the `mutable` keyword
• Use as few global objects as possible
• If you must use a global object, provide only `const`-qualified interface
• Don’t use output arguments

All of these are good ideas for single-threaded code. You can do this now!