Task Chaining for Easier Threaded Callbacks

Dr Christopher Jones

HSF Framework Meeting

8 September 2021

Overview

- Tasks and callbacks
- Chaining tasks

Task Working Definition

- A unit of work to be processed on one thread

TBB Tasks

- TBB has the abstract notion of a task but no task objects

- Tasks are controlled by **`tbb::task_group`**
  - tasks are associate to a `tbb::task_group`
  - a task is queued to be run later by calling ***`tbb::task_group`*`::run`**
    - queued tasks can be run on any available TBB controlled thread
    - one can call `tbb::task_group::run` from within a running task
  - ***`tbb::task_group`*`::wait`** returns only after all tasks in a group have finished

TBB Task Example

- Task ordering
  - ***a*** can run concurrently with
    - ***b***, ***c*** and ***d***
  - ***b*** must run before ***c*** and ***d***
  - ***c*** can run concurrently with ***d***

```
tbb::task_group g;

g.run([]() { a(); });

g.run([&g]() {
     b();
     g.run([]() { c(); });
     g.run([]() { d(); });
});

g.wait();
```

CMS Task

- Built on top of `tbb::task_group`

- **`WaitingTask<>`** represents a task
  - Holds onto the functor which does the work
    - functor takes `std::exception_ptr` as argument to propagate exception from task to task
  - Has an internal reference count
    - used to implement the same callback assigned to multiple other tasks
  - Call `make_waiting_task` with lambda to create an an instance

- **`WaitingTaskHolder`**
  - Associates `tbb::task_group` to a `WaitingTask`
  - Handles reference count
    - when count goes to 0 the holder causes `tbb::task_group::run` to be called

CMS Task Example

- Task ordering
  - ***a*** can run concurrently with
    - ***b***, ***c*** and ***d***
  - ***b*** must run before ***c*** and ***d***
  - ***c*** can run concurrently with ***d***

```
using Holder = WaitingTaskHolder;
tbb::task_group g;

Holder(g, make_waiting_task([](auto e)
                              { a(); }));

Holder(g, make_waiting_task([&g](auto e) {
    b();
    Holder(g, make_waiting_task([](auto e)
                                  { c(); }));
    Holder(g, make_waiting_task([](auto e)
                                  { d(); })); })

);
g.wait();
```

note: all examples have

```
using namespace edm;
using namespace edm::waiting_task::chain:
```

CMS Task Exception Handling

- Exceptions have to be propagated manually

```
void async_a(WaitingTaskHolder nextTask) {

  WaitingTaskHolder h_a(nextTask.group(),
                        make_waiting_task([nextTask](auto e) {
    if(not e) {
      try { a(); } catch(…) {
          nextTask.doneWaiting(std::current_exception());
      }
    } else { nextTask.doneWaiting(e); }
  });
  async_before_a(h_a); //h_a will be called after other tasks
}
```

CMS Task Exception Handling

- Exceptions have to be propagated manually

```
void async_a(WaitingTaskHolder nextTask) {

  WaitingTaskHolder h_a(nextTask.group(),
                        make_waiting_task([nextTask](auto e) {
    if(not e) {
      try { a(); } catch(…) {
          nextTask.doneWaiting(std::current_exception());
      }
    } else { nextTask.doneWaiting(e); }
  });
  async_before_a(h_a); //h_a will be called after other tasks
}
```

CMS Task Exception Handling

- Exceptions have to be propagated manually

```
void async_a(WaitingTaskHolder nextTask) {

  WaitingTaskHolder h_a(nextTask.group(),
                        make_waiting_task([nextTask](auto e) {
    if(not e) {
      try { a(); } catch(…) {
          nextTask.doneWaiting(std::current_exception());
      }
    } else { nextTask.doneWaiting(e); }
  });
  async_before_a(h_a); //h_a will be called after other tasks
}
```

Task Chaining

- It is useful to be able to create a Task from within another Task
  - Provides a way to synchronize work across threads
  - Requires knowing at compile time task dependencies

- Callbacks
  - Can be useful to pass a Task to other Tasks
  - Once all owning Tasks finish the callback Task is run
  - Allows dynamic scheduling of Tasks
    - e.g. scheduling of framework modules based on data dependencies found at run time

- The downside to chains of callbacks …

“Backwards call you must”

Backward calls

- Want the order
  - ***a_async***
  - ***b_async***
  - ***c***

```
tbb::task_group g;

FinalWaitingTask t_f;

auto t_c = make_waiting_task(
    [h = Holder(g, t_f)](auto) { c(); } );

auto t_b = make_waiting_task(
    [h = Holder(g, t_c)](auto) { b_async(move(h)); });

a_async(Holder(g, t_b));
g.wait();  throw_if(t_f.exception());
```

Simplifying the Syntax

- Want to specify tasks in the order we'd like them to run
- Want to have exceptions propagate automatically
  - with option to do it explicitly

- Use unix pipe metaphor
  - similar to C++ 20 range library
  - basic functions
    - **`first`** : starts off the chain
    - **`then`** : extends chain
    - **`ifThen`** : conditionally extends chain
    - **`runLast`** : ends chain and runs first task on this thread
    - **`lastTask`**: ends chain and creates a WaitingTaskHolder with first task

Chained Calls

- Want the order
  - ***a_async***
  - ***b_async***
  - ***c***

```
tbb::task_group g;

FinalWaitingTask t_f;

first( [](auto nextTask) { a_async(nextTask); })
| then([](auto nextTask) { b_async(nextTask); })
| then([](auto nextTask) { c(); })
| runLast(Holder(g, t_f));

g.wait(); throw_if(t_f.exception());
```

Chains Are Composable

- Use chains to write functions which can be called from a chain

```
void b_async(Holder waitTask) {
  first( [](auto nextTask) { d_async(nextTask); })
  | then([](auto nextTask) { e_async(nextTask); })
  | runLast(waitTask);
}
```

Chains Are Composable

- Can use chains inside chains

```
vector<double> v(foos.size(),0.);
atomic<double> sum(0.);

first([&](auto nextTask) {
 for(int i=0; i< foos.size(); ++i) {
    first([&v, &foos, i](auto nt) { v[i]=foos[i].run();} )
    | lastTask(nextTask); //queues task instead of running it
 }})
| then([&](auto nextTask) { for(auto val: v) { sum +=v;} })
| then([&](auto nextTask) { final_async(sum, nextTask); })
| runLast(waitTask);
```

Conditionally Running a Task

- Can avoid running a task
  - must know at task chaining call if can skip

```
first(…)
| …
| ifThen(not foos.empty(),
          [&v](auto nextTask) {
      for(auto&& f: foos) {e_async(nextTask, f); } })
|…
```

Exception Handling

- Exception propagated between tasks automatically for simple case
  - when lambda used has just one argument
  - tasks following after a task that threw an exception will not be run

- Can explicitly handle exception propagation
  - use lambda with two arguments

```
[](Holder nextTask, std::exception_ptr const& ep) {
   if(not ep) {
      try { a(); } catch(…) {
          nextTask.doneWaiting(std::current_exception());
      }
    } else { nextTask.doneWaiting(ep); }
};
```

Exception Handling

- Exception propagated between tasks automatically for simple case
  - when lambda used has just one argument
  - tasks following after a task that threw an exception will not be run

- Can explicitly handle exception propagation
  - use lambda with two arguments

```
[](Holder nextTask, std::exception_ptr const& ep) {
   if(not ep) {
      try { a(); } catch(…) {
          nextTask.doneWaiting(std::current_exception());
      }
    } else { nextTask.doneWaiting(ep); }
};
```

Exception Handling

- Exception propagated between tasks automatically for simple case
  - when lambda used has just one argument
  - tasks following after a task that threw an exception will not be run

- Can explicitly handle exception propagation
  - use lambda with two arguments

```
[](Holder nextTask, std::exception_ptr const& ep) {
   if(not ep) {
      try { a(); } catch(…) {
          nextTask.doneWaiting(std::current_exception());
      }
    } else { nextTask.doneWaiting(ep); }
};
```

Exception Handling (continued)

- Can use helper to skip task and handle exception manually
  - ifException.else_

```
| then(ifException([](std::exception_ptr const& ep) {
          printException(ep); })
       .else_([](auto nextTask) {
          do_work_async(nextTask);
       })
     )
| …
```

Exception Handling (continued)

- Can use helper to skip task and handle exception manually
  - ifException.else_

```
| then(ifException([](std::exception_ptr const& ep) {
          printException(ep); })
       .else_([](auto nextTask) {
          do_work_async(nextTask);
       })
      )
| …
```

Exception Handling (continued)

- Can use helper to skip task and handle exception manually
  - ifException.else_

```
| then(ifException([](std::exception_ptr const& ep) {
          printException(ep); })
       .else_([](auto nextTask) {
          do_work_async(nextTask);
       })
      )
| …
```

Code

- Documentation
  - https://github.com/cms-sw/cmssw/blob/master/FWCore/Concurrency/README.md

- Code can be found here
  - https://github.com/cms-sw/cmssw/blob/master/FWCore/Concurrency/interface/chain_first.h