Compose Chaos

Part 1: RAII Applied to Concurrency

<image of “chaos” in a lego piece shape>

A. Joël Lamotte

- Experience:
  Games, Tools, Embedded, Mobile, Web...
- **C++**, Python, C#, JavaScript, Java, ActionScript...
- Today: **SoftBank Robotics Europe**
  (**libqi**, code “expert”, etc.)

WE ARE RECRUITING!

Are you familiar with:

- mutex/locks
- atomics
- futures/promises, continuations, “.then()”
- “async(task)” -like functions
- concurrent containers
- executors
- thread-pools
- strands

This talk is part of an ongoing exploration...

- ~~“expert friendly”~~
- ~~Parallelization~~
- “Heterogeneous” Concurrency
- Mostly pseudo code
- Note **slides numbers** for questions at the end

“Heterogeneous” Concurrency

Different kinds of:

- concurrency
  - including “fork-join” parallelization
- data
- processing resources

Must work together in the same application

Most important rule when writing concurrent code:

Do not share data.

If you can.

You can't.

Principles/Hypotheses

1. RAII applied to concurrency
2. No raw synchronization primitives
3. Task ⇔ message
4. System ⇔ “Actor-like”
5. 1 synchronization mechanism per abstraction
6. `Executor( Task{} );`

Principles/Hypotheses

1. **RAII applied to concurrency**
2. No raw synchronization primitives
3. Task ⇔ message
4. System ⇔ “Actor-like”
5. 1 synchronization mechanism per abstraction
6. `Executor( Task{} );`

RAII applied to concurrency

Hypotheses:

- Constructor and destructor are **synchronization points**.
- Tasks launched by an object and accessing internal data of this object are **part of that object's lifetime**.

RAII: Constructor and Destructor

1. Constructor: once constructed, an object must be able to handle concurrent member calls.

2. Destructor: all tasks launched by this object and accessing its internal data must be synchronized before the end of the destructor.

RAII: Constructor and Destructor

1. Constructor: once constructed, an object must be able to handle concurrent member calls.

2. Destructor: all tasks launched by this object and accessing its internal data must be synchronized before the end of the destructor.

```cpp
struct Data { int code; /* ... */ };

// might be concurrent
future<Data> crunch(int value);

struct System {
 explicit System(int x)
   : store{ crunch(x).get() }
 {}
 // ...
private:
 vector<Data> store;
 // ...
};
```

```cpp
future<vector<int>> big_fat_random_garbage();

struct System {
 explicit System(int x) {
   auto ft_1 = crunch(some_cosmic_constant);
   auto ft_2 = crunch(x);
   auto ft_ints = big_fat_random_garbage();

   store = {ft_1.get(), ft_2.get()};
   values = ft_ints.get();
 }
 // ...
private:
 vector<Data> store;
 vector<int> values;
 // ...
};
```

```
int math_ninja_flip(const Data& x, const Data& y, int value);

struct System {
 explicit System(int x) {
   auto ft_1 = crunch(some_cosmic_constant);
   auto ft_2 = crunch(x);
   store = {ft_1.get(), ft_2.get()};

   values = { math_ninja_flip(store[0], store[1], x) };
 }
 // ...
private:
 vector<Data> store;
 vector<int> values;
 // ...
};
```

```cpp
int math_ninja_flip(const Data& x, const Data& y, int value);

struct System {
 explicit System(int x)
   : store{crunch(some_cosmic_constant).get(), crunch(x).get()}
   , values{ math_ninja_flip(store[0], store[1], x) }
 {
 }
 // ...
private:
 vector<Data> store;
 vector<int> values;
 // ...
};
```

RAII: Constructor

After the end of the constructor, the object must be able to handle concurrent member calls.

- Initialization must be synchronous, even if blocking is necessary (!!)
  - Don't Block™?

```cpp
struct System {
 explicit System(int x) {
   auto ft_data2 = crunch(some_cosmic_constant);
   crunch(x).then([=](auto ft_data){
     store.push_back( ft_data.get() );
   });

   big_fat_random_garbage().then([&](auto ft_ints){
     values = ft_ints.get();
   });
 }
 // ...
private:
 vector<Data> store;
 vector<int> values;
 // ...
};
```

RAII: Constructor

After the end of the constructor, the object must be able to handle concurrent member calls.

- Initialization must be synchronous
- Do not defer initialization after constructor’s end

RAII: Constructor

After the end of the constructor, the object must be able to handle concurrent member calls.

- Initialization must be synchronous
- Do not defer initialization after constructor's end?

```
struct System {
 explicit System(int x) {
   auto ft_1 = crunch(some_cosmic_constant);  // Takes a long time...
   auto ft_2 = crunch(x);    // Takes a long time...
   auto ft_ints = big_fat_random_garbage(); // Takes a long time...

   store = {ft_1.get(), ft_2.get()}; // Wait for a long time...
   values = ft_ints.get(); // Might wait a long time...
 }

private:
 vector<Data> store;
 vector<int> values;
 // ...
};
```

RAII: Constructor

After the end of the constructor, the object must be able to handle concurrent member calls.

- Initialization must be synchronous
- ~~Do not defer initialization after constructor’s end...?~~
- Exit the constructor as soon as ready to handle concurrent member calls?

```cpp
struct System {
 explicit System(int x) {
   crunch(x).then([&]( auto ft_data ){ // *
     scoped_lock lock{this_mutex};
     store.push_back( ft_data.get() );
   });
   big_fat_random_garbage()
     .then([&](auto ft_ints){ // *
       scoped_lock lock{this_mutex};
       values.push_back( ft_ints.get() );
     });
 }
 // ...
private:
 vector<Data> store;
 vector<int> values;
 mutex this_mutex;
 // ...
};
```

```cpp
future<int> System::foowizz(Data data) {
   return async( [this, data]{ // *
     scoped_lock lock{this_mutex};
     int result = math_ninja_flip(
          store.back(), data, values.back());
     store.push_back(data);
     values.push_back(data.code);
     return result;
   });
}

// * : Tasks are not garanteed to be executed
// at a time where the object is still alive!
// Also initialization may finish after
// other tasks are executed!
```

```
System sys_a{1234};
sys_a.foowizz(some_data);

System sys_b{23423};
sys_b.foowizz(some_data);

System sys_c{4329423};
sys_c.foowizz(some_data);
```

```cpp
System sys_a{1234};
System sys_b{23423};
System sys_c{4329423};
// "Simplified" version
auto ft_data =
  sys_a.foowizz(some_data)
  .then([](int value){
    return crunch(value);
  })
  .then([&](Data data){
    return sys_b.foowizz(data);
  })
  .then([](int value){
    return crunch(value);
  })
  }.then([&](Data data){
    return sys_c.foowizz(data);
  });
```

RAII: Constructor

After the end of the constructor, the object must be able to handle concurrent member calls.

- Initialization must be synchronous
- Do not defer initialization after constructor’s end
- Optimization: Exit the constructor as soon as ready to handle concurrent member calls

```cpp
struct System {
 explicit System(int x) {
   auto ft_data = crunch(x);
   auto ft_ints = big_fat_random_garbage();
   work_queue.push( [=]{
     store.push_back(ft_data.get());
     values = ft_ints.get();
   });
 }
 future<int> foowizz(Data data); // ...
private:
 vector<Data> store;
 vector<int> values;
 concurrent_queue<function<void()>>
    work_queue;
 thread update_thread{ update_loop };
 void update_loop(); // pop and execute tasks
};
```

```cpp
future<int> System::foowizz(Data data) {
  return work_queue.push( [this, data]{
    int result = math_ninja_flip(
      store.back(), data,
      values.back());
    store.push_back(data);
    values.push_back(data.code);
    return result;
  });
 }
```

```cpp
struct System {
 explicit System(int x) {
   auto ft_data = crunch(x);
   auto ft_ints = big_fat_random_garbage();
   strand.push( [=]{
     store.push_back(ft_data.get());
     values = ft_ints.get();
   });
 }
 future<int> foowizz(Data data); // ...
private:
 vector<Data> store;
 vector<int> values;
 Strand strand;
 // ...
};
```

```cpp
 future<int> System::foowizz(Data data) {
   return strand.push( [this, data]{
     int result = math_ninja_flip(
store.back(), data, values.back());
     store.push_back(data);
     values.push_back(data.code);
     return result;
   });
  }
```

RAII: Constructor and Destructor

1. Constructor: once constructed, object must be able to handle concurrent member calls.

2. Destructor: all tasks launched by this object and accessing its internal data must be synchronized before the end of the destructor.

RAII: Destructor

Before the end of the destructor, all tasks launched accessing internal data must **either**:

- **Be finished**
- **Do nothing on execution**
    - Don't throw!

RAII: `Destructor`

- `Wait for tasks to finish`

```cpp
class Poppy {

 template<class F>
 void launch(F task) {
   launched_tasks.push(async(executor, task));
 }

public:
 Poppy(Executor exec) : executor(exec) {
    launch([]{ init_background_system(); });
 }

 ~Poppy() {
   future<void> ft;
   while(launched_task.pop(ft))
       ft.wait();
 }
```

```cpp
  int count() const { return counter; }

  void im_poppy(){
    launch([&]{
      some_dirty_laundry();
      ++counter;
    });

private:
 atomic<int> counter;
 Executor executor;
 concurrent_queue<future<void>>
   launched_tasks;
};
```

```cpp
class Poppy {
 //...
 ~Poppy() {
   future<void> ft;
   while(launched_task.pop(ft))
       ft.wait();
 } // ...
};
```

```cpp
class BigSystem {
 ThreadPool thread_pool;
 Poppy poppy{ &thread_pool };
 // ...
public:
 ~BigSystem() {
   thread_pool.stop_tasks_execution();
   some_cleanup();
 } // ~Poppy() might be blocked!

 void be_funny() { poppy.im_poppy(); }
 // ...
};
```

```cpp
class Poppy {
 //...
 ~Poppy() {
  future<void> ft;
  while(
    launched_task.pop(ft)
  ) ft.wait();
 } // ...
};
```

```cpp
void be_poppy_for_some_time(const seconds timeout){
  concurrent_queue<function<void()>> work_queue;
  Poppy poppy{ [&](auto task){
    work_queue.push(task);
  }};

  auto begin_time = some_clock::now();
  begin_try_to_be_poppy(poppy); // calls poppy's members a lot
  while((some_clock::now() - begin_time) < timeout) {
    function<void()> task;
    if(work_queue.pop(task))
      task();
  } // No more task execution
  end_try_to_be_poppy(poppy);
} // ~Poppy() might be blocked!
```

```cpp
class Poppy {
 //...
 ~Poppy() {
  future<void> ft;
  while(
    launched_task.pop(ft)
  ) ft.wait();
 } // ...
};
```

```cpp
void be_poppy_for_some_time(const seconds timeout){
  Strand strand;
  Poppy poppy{ &strand };

  begin_try_to_be_poppy(poppy); // calls poppy's members a lot
  sleep_for(timeout);
  strand.join(); // stop executing tasks, MAYBE destroy tasks?
  end_try_to_be_poppy(poppy);
} // ~Poppy() might be blocked!
  // Depends on what the executor guarantees...
```

RAII: `Destructor`

- ~~`Wait for tasks to finish`~~
- `Notify tasks to do nothing on execution`

```cpp
class Poppy {
  template<class F> void launch(F f) {
    async(executor, [=]{
      if(tasks_must_not_execute) // might not exist!
        f();
    });
  }
public:
 ~Poppy() {
   tasks_must_not_execute = true;
   // don't wait for running tasks...
 }
  void im_poppy() {
    launch([this]{
      some_dirty_laundry();
      ++counter; // might not exist!
    });
  }
```

```cpp
// ...
private:
  atomic<int> counter{0};
  atomic<bool>
    tasks_must_not_execute{false};
 // ...
};
```

```cpp
class Poppy {
  template<class F> void launch(F f) {
    weak_ptr maybe_state{state}; // not thread-safe!
    async(executor, [=]{
      if(auto state = maybe_state.lock())
        f(state);
    });
  }
public:
  ~System() = default; // no wait

  void im_poppy(){
    launch([](auto state){
      some_dirty_laundry();
      ++state->counter; // forced "living" after destruction
    });
  }
 // ...
```

```cpp
private:
 struct State {
   atomic<int> counter{0};
 };

 // needs protection too!
 shared_ptr<State> state =
   make_shared<State>();

 // ...
};
```

RAII: Destructor

- ~~Wait for tasks to finish~~
- ~~Notify tasks to do nothing on execution~~
- Wait running tasks to finish?

RAII: Destructor - The Right Way*

1. Notify all tasks to “join”;
    - If a task did not execute yet, make it do nothing on execution;
2. Wait already running tasks to finish;
    - Stop waiting as soon as you can ignore tasks’ end;

* Except in critical cases where you have to execute all tasks anyway: banking, critical logging, black boxes systems for exemple.

RAII: Destructor

1. Notify **all tasks** to “join”;
   - If a task did not execute yet, make it do nothing on execution;
2. Wait already running tasks to finish;
   - Stop waiting as soon as you can ignore tasks end;

Keep track of internal tasks’ lifetime

RAII: `Destructor`

1. `Notify all tasks to “join”;`
   - `If a task did not execute yet, make it do nothing on execution;`
2. `Wait already` **`running tasks`** `to finish;`
   - `Stop waiting as soon as you can ignore tasks end;`

`Keep track of internal tasks’ status`

RAII: Destructor

1. **Notify** all tasks to “join”;
   - If a task did not execute yet, make it do nothing on execution;
2. **Wait** already running tasks to finish;
   - Stop waiting as soon as you can ignore tasks end;

Use a higher-level task synchronization tool (could be a strand or something similar...)

```cpp
class Poppy {
  template<class F> void launch(F task) {
    auto synchronized_task = task_sync.synched(task);
    async(executor, synchronized_task);
  }
public:
  ~System() = default;
  // task_sync.join() called in ~TaskSynchronizer()
  // wait for running tasks, deactivate all other tasks

  void im_poppy(){
    launch([&]{
      some_dirty_laundry();
      ++counter;  // 'this' is garanteed to be alive
                  // if this code is executing
    });
  }
 // ...
```

```cpp

private:
  atomic<int> counter;
  // ...
  TaskSynchronizer task_sync;
};
```

RAII: Destructor

1. Notify all tasks to “join”;
   - If a task did not execute yet, make it do nothing on execution;
2. Wait already running tasks to finish;
   - Stop waiting as soon as you can ignore tasks end;
3. Do not wait asynchronous calls from the destructor: fire and forget?

RAII: Destructor

1. Notify all tasks to “join”;
    - If a task did not execute yet, make it do nothing on execution;
2. Wait already running tasks to finish;
    - Stop waiting as soon as you can ignore tasks end;
3. Do not wait asynchronous calls from the destructor: ~~fire and forget.~~ If reasonable?

RAII: Destructor

Do not wait asynchronous calls from the destructor: fire and forget, if reasonable.

```
struct System {
 // ...
 World& world; // dependency, not owned
public:
 System(World& w) : world(w) {}
 ~System()
 {
   world.notify_everybody(SystemEnd{}).get();
 }
 // ...
};
```

```
struct World {
 // ...
 template<class M>
 future<void> notify_everybody(M message);
};
```

RAII: Destructor

Do not wait asynchronous calls from the destructor: fire and forget, If reasonable.

```
struct System {
 // ...
 World& world; // dependency, not owned
public:
 System(World& w) : world(w) {}
 ~System()
 {
   world.notify_everybody(SystemEnd{});
   // fire and forget: ok
 }
 // ...
};
```

```
struct World {
 // ...
 template<class M>
 future<void> notify_everybody(M message);
};
```

RAII: Destructor

Do not wait asynchronous calls from the destructor: fire and forget, if reasonable.

```
class Focus { // Unique focus acquisition
 SomeSystem& system;
public:
 // ... move-only
 Focus(SomeSystem& sys)
   : system{ sys }
 { system.acquire_focus().get(); } // wait to get it
 // fire and forget: might be ok, we don't really own or share the resource
 ~Focus() { system.notify_focus_end_use(); } // async call, no wait
 // ...
};
```

RAII: `Constructor and Destructor`

`Observations:`

```cpp
template<class T> class Mechanism { /* ...
*/ };

class SubSystem {
 BigData data;
 Mechanism cog{data};
 // ...
};

class BigSystem {
 ThreadPool executor;
 SubSystem a{executor};
 OtherSubSystem b{a, executor};
 WeirdSubSystem c{b, executor};
 // ...
};
```

```cpp
class VeryBigSystem
{
 BigSystem k;
 OtherBigSystem l;
 TroublingSystem m{l, k};
 // ...
};
```

```cpp
class Client
{
public:
   Client( std::string window_name, Configuration
configuration );
   ~Client() = default;

   core::TaskChainThread io_executor;

   core::SystemIO system_io{ io_executor };
   core::WindowSystem window_io{ system_io };
   core::Window main_window;

   input::InputEngine input_engine;
   graphic::GraphicEngine graphic_engine;
   GraphicComponents graphic_components;
   ActionsRegistry actions_registry;
   // ...
};
```

```cpp
Client::Client( std::string window_name,
Configuration config )
     : io_executor( IO_CYCLE_FREQUENCY )
     , main_window( window_io, window_name,
         core::read_window_state( config ) )
     , input_engine{ system_io, main_window,
         input_config( config ) }
     , graphic_engine{ std::ref(io_executor),
         main_window, config }
     , graphic_components{ graphic_engine }
     , actions_registry{ input_engine }
 {
     main_window.show();
 }
```

RAII: `Constructor and Destructor`

`Observations:`

- `Like for all RAII-based code, reasoning concurrent code appears “easy”, familiar, unsurprising.`
- `Helps with correctness.`
- `Opens door to potential optimizations (by avoiding blocking internally once algorithms are correct).`

Principles/Hypotheses

1. RAII applied to concurrency
2. No raw synchronization primitives
3. Task ⇔ message
4. System ⇔ “Actor-like”
5. 1 synchronization mechanism per abstraction
6. `Executor( Task{} );`

RAII applied to Concurrency

Questions?

References:
Sean Parent's concurrency talk: https://www.youtube.com/watch?v=32f6JrQPV8c
Book "C++ Concurrency In Action":
https://www.manning.com/books/c-plus-plus-concurrency-in-action

@mjklaim mjklaim@gmail.com or jlamotte@softbankrobotics.com

RAII applied to Concurrency

BONUS CODE!

`TaskSynchronizer`

Projects & Experimentations

My Technical Goals:

1. code maintainable for **long-term**
2. **scalable** usage of available resources
3. good **performance**
4. open door to easy **extensibility**

Projets & Experimentations

Qi library (SBE): https://github.com/aldebaran/libqi

An inter-process RPC-based communication middleware and concurrency coding tools.

- Targets robotics use cases (multi-process)
- Gives good perspective vs experimentations

WE ARE STILL RECRUITING