



Multi-threaded *art* forum – discussing modules

Kyle J. Knoepfel

1 December 2017

For today

- Quick review from last time
- State transitions and schedules
- Shared modules
- Replicated modules
- Next steps

Last time

https://cdcvs.fnal.gov/redmine/projects/art/wiki/Art_multi-threading_forum_-_introduction

- We discussed motivations for a multi-threaded (MT) framework
- Largely based off of CMSSW's design
 - We use Intel's Threading Building Blocks (TBB)
 - Steps to be performed are factorized into *tasks*
 - You can think of a call to your module's "produce" function as performing a task
- Users specify the number of concurrent schedules (i.e. event loops) and (optionally) the maximum number of threads that the process can use.
- Each loop processes one event at a time.
- Different modules will also be able to be run in parallel on the *same* event.

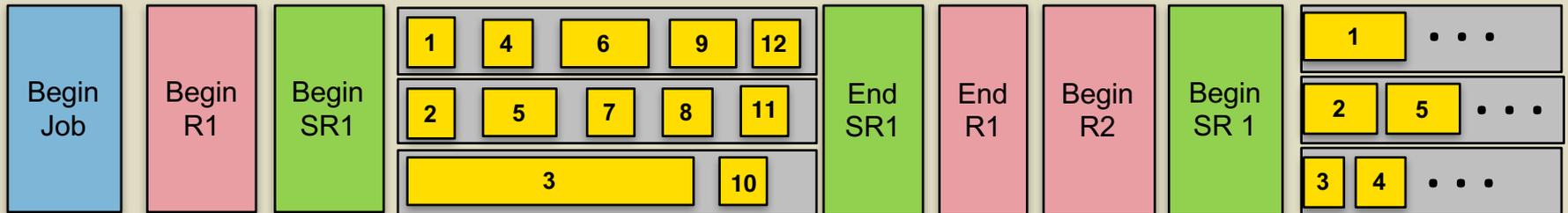
Last time

https://cdcvs.fnal.gov/redmine/projects/art/wiki/Art_multi-threading_forum_-_introduction

- We discussed motivations for a multi-threaded (MT) framework
- Largely based off of CMSSW's design
 - We use Intel's Threading Building Blocks (TBB)
 - Steps to be performed are factorized into *tasks*

You can think of a call to your module's "produce" function as performing a task

Currently implemented:



Last time

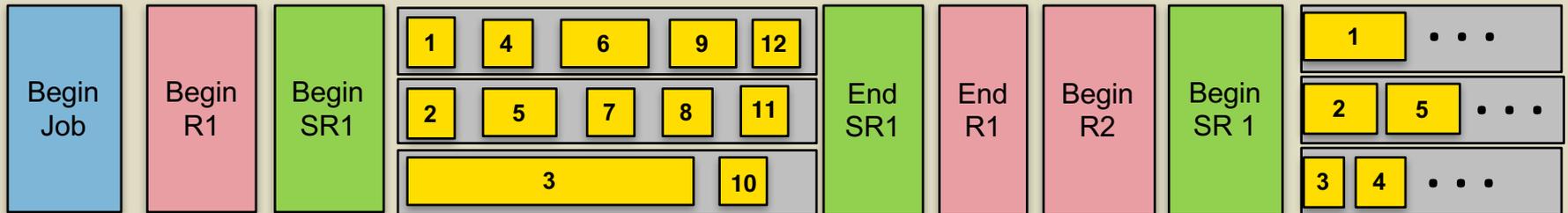
https://cdcvs.fnal.gov/redmine/projects/art/wiki/Art_multi-threading_forum_-_introduction

- We discussed motivations for a multi-threaded (MT) framework
- Largely based off of CMSSW's design
 - We use Intel's Threading Building Blocks (TBB)
 - Steps to be performed are factorized into *tasks*

You can think of a call to your module's "produce" function as performing a task

Currently implemented:

But what about the modules? Today's topic.



- *art* guarantees that any currently-existing modules (to within some interface changes) will be usable in a multi-threaded execution of *art*.
 - No multi-threading benefits will be realized with such “legacy” modules
- To take advantage of *art*'s multi-threading capabilities, users will need to choose the kind of module they use:
 - **Serialized module**: use if the facilities you are using do not allow for concurrent execution and you must see all events
 - **Per-event loop module**: for a configured module, one copy of that module is produced per event loop—each module copy sees one event at a time. Use if moving to a fully concurrent module is unfeasible
 - **Fully concurrent module**: module functions can be called concurrently without any data races

General statements—modules

- *art* guarantees that any currently-existing modules (to within some interface changes) will be usable in a multi-threaded execution of *art*.
 - No multi-threading benefits will be realized with such “legacy” modules
- To take advantage of *art*'s multi-threading capabilities, users will need to choose the kind of module they use:

- **Serialized module:** use if the facilities you are using do not allow for concurrent execution and you must see all events
 - **Per-event loop module:** for a configured module, one copy of that module is produced per event loop—each module can only see one event at a time. Use if moving to a fully concurrent module is unfeasible
 - **Fully concurrent module:** module functions can be called concurrently without any data races
- Changed to something simpler...*

General statements—modules

- *art* guarantees that any currently-existing modules (to within some interface changes) will be usable in a multi-threaded execution of *art*.
 - No multi-threading benefits will be realized with such “legacy” modules
- To take advantage of *art*'s multi-threading capabilities, users will need to choose the kind of module they use:
 - **Shared module:** sees all events—calls can be serialized or asynchronous.
 - **Replicated module:** for a configured module, one copy of that module is created per schedule—each module copy sees one event at a time. Use if moving to a concurrent, shared module is not feasible.

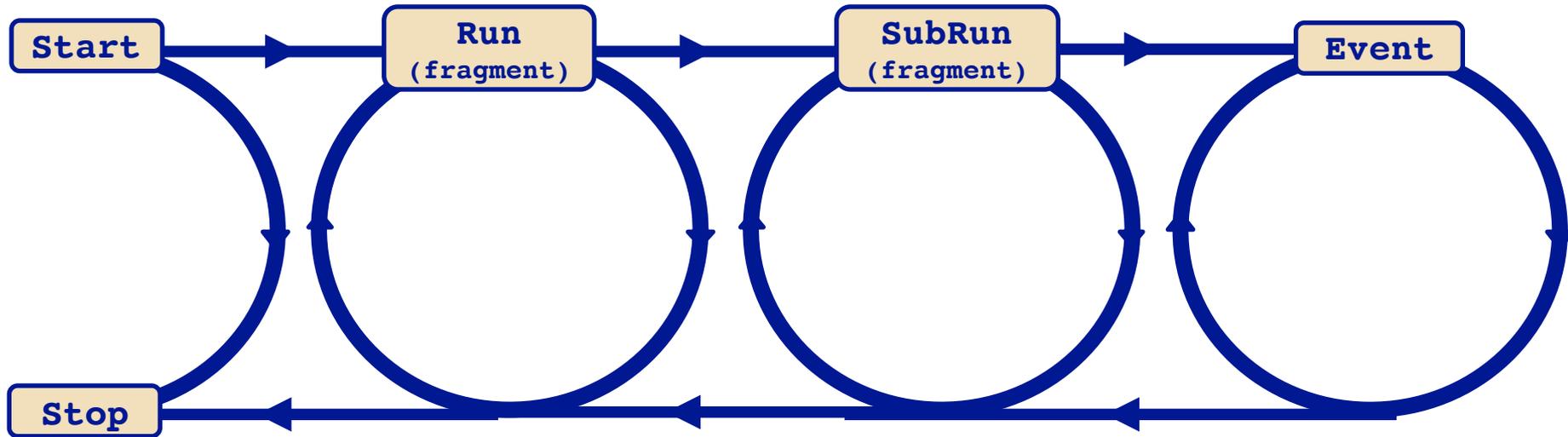
What you see and what you don't see in *art* or State transitions and schedules

Allowed transitions

- *art* is designed to process a hierarchy of data-containment levels:
 - *Run* \supset *SubRun* \supset *Event*
- *art* users expect the framework to respect this hierarchy

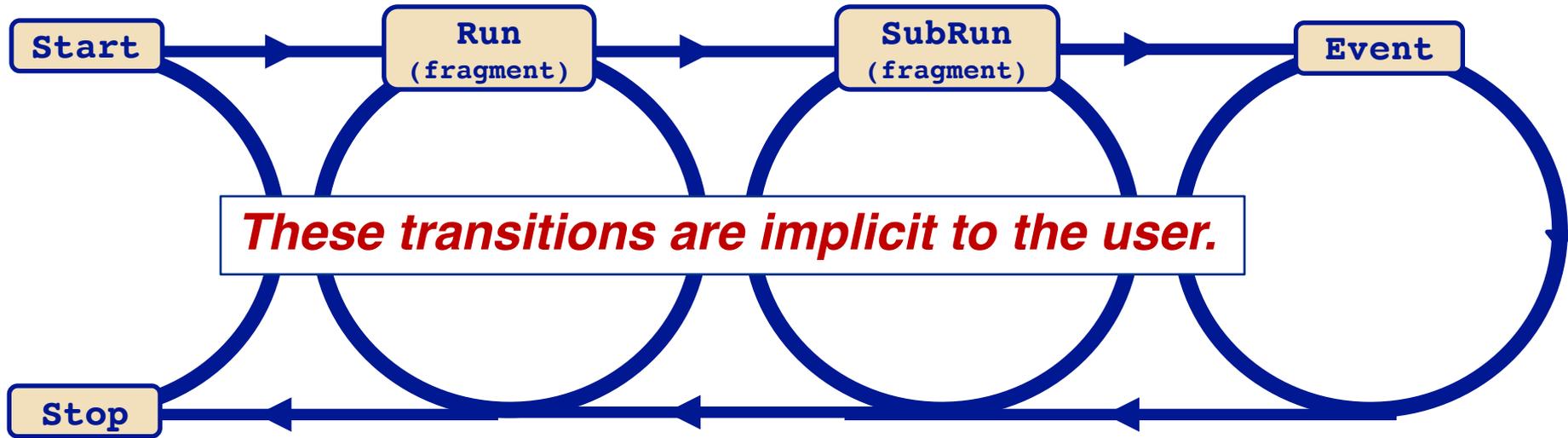
Allowed transitions

- *art* is designed to process a hierarchy of data-containment levels:
 - *Run* \supset *SubRun* \supset *Event*
- *art* users expect the framework to respect this hierarchy
- The allowed transitions by the framework are thus:



Allowed transitions

- *art* is designed to process a hierarchy of data-containment levels:
 - *Run* \supset *SubRun* \supset *Event*
- *art* users expect the framework to respect this hierarchy
- The allowed transitions by the framework are thus:



Processing a data-containment level (e.g. Event)

- The order in which modules are executed for a Run, SubRun, or Event is determined by the **path declarations** in the configuration file.

```
physics: {  
  
  producers: {  
    makeHits: {...}  
    makeShowers: {...}  
    produceG4Steps: {...}  
  }  
  analyzers: {  
    plotHits: {...}  
  }  
  
  hitPath: [makeHits, makeShowers]  
  geomPath: [produceG4Steps]  
  analyzePath: [plotHits]  
}
```

Module declarations

Path declarations

Processing a data-containment level (e.g. Event)

- The order in which modules are executed for a Run, SubRun, or Event is determined by the **path declarations** in the configuration file.

```
physics: {  
  producers: {  
    makeHits: {...}  
    makeShowers: {...}  
    produceG4Steps: {...}  
  }  
  analyzers: {  
    plotHits: {...}  
  }  
}
```

Trigger path	hitPath: [makeHits, makeShowers]
Trigger path	geomPath: [produceG4Steps]
End path	analyzePath: [plotHits]

Processing a data-containment level (e.g. Event)

- The order in which modules are executed for a Run, SubRun, or Event is determined by the **path declarations** in the configuration file.

```
physics: {  
  producers: {  
    makeHits: {...}  
    makeShowers: {...}  
    produceG4Steps: {...}  
  }  
  analyzers: {  
    plotHits: {...}  
  }  
}
```

Trigger path	hitPath: [makeHits, makeShowers]
Trigger path	geomPath: [produceG4Steps]
End path	analyzePath: [plotHits]

- The order in which *trigger paths* are executed is unspecified (current *art*).
- In MT *art* trigger paths will be executed simultaneously.
- Modules in a trigger path are executed in the order specified.
- End paths are always processed after all trigger paths.

Processing a data-containment level (e.g. Event)

- The order in which modules are executed for a Run, SubRun, or Event is determined by the **path declarations** in the configuration file.

```
physics: {
```

- The order in which *trigger paths are executed* is

- It is the **schedule** that is responsible for calling modules in the correct order.
- Diagramming the interaction between the state transitions and the schedule is non-trivial, especially in a multi-threaded environment.
- However, we will try to do this with **three** modules: **m1**, **m2**, and **m3**.

Trigger path	hitPath: [makeHits, makeShowers]
Trigger path	geomPath: [produceG4Steps]
End path	analyzePath: [plotHits]
	}

executed in the order specified.

- End paths are always processed after all trigger paths.

Time structure for calling modules

Single schedule (current *art*)



Time structure for calling modules

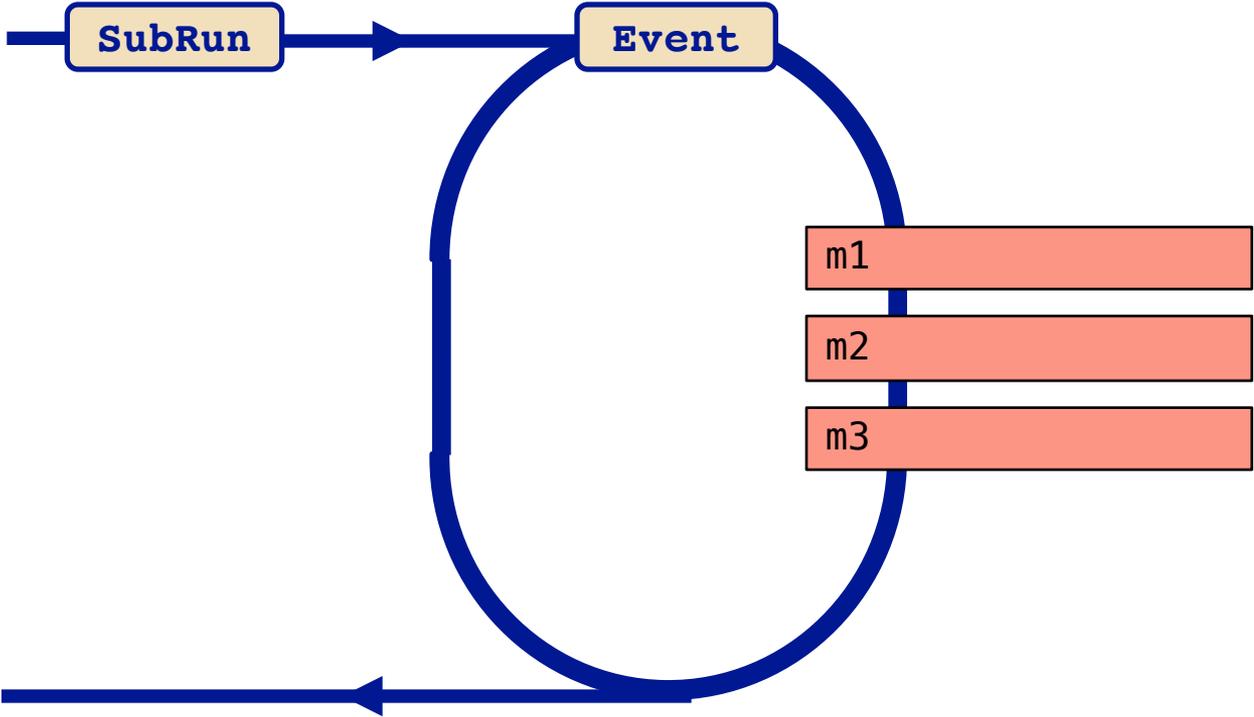
Single schedule—current *art*

Begin
SR1

1 2

3

End
SR1

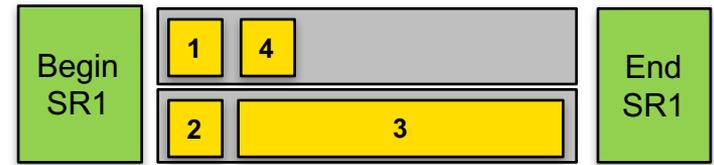


Shared modules

Modules shared across schedules

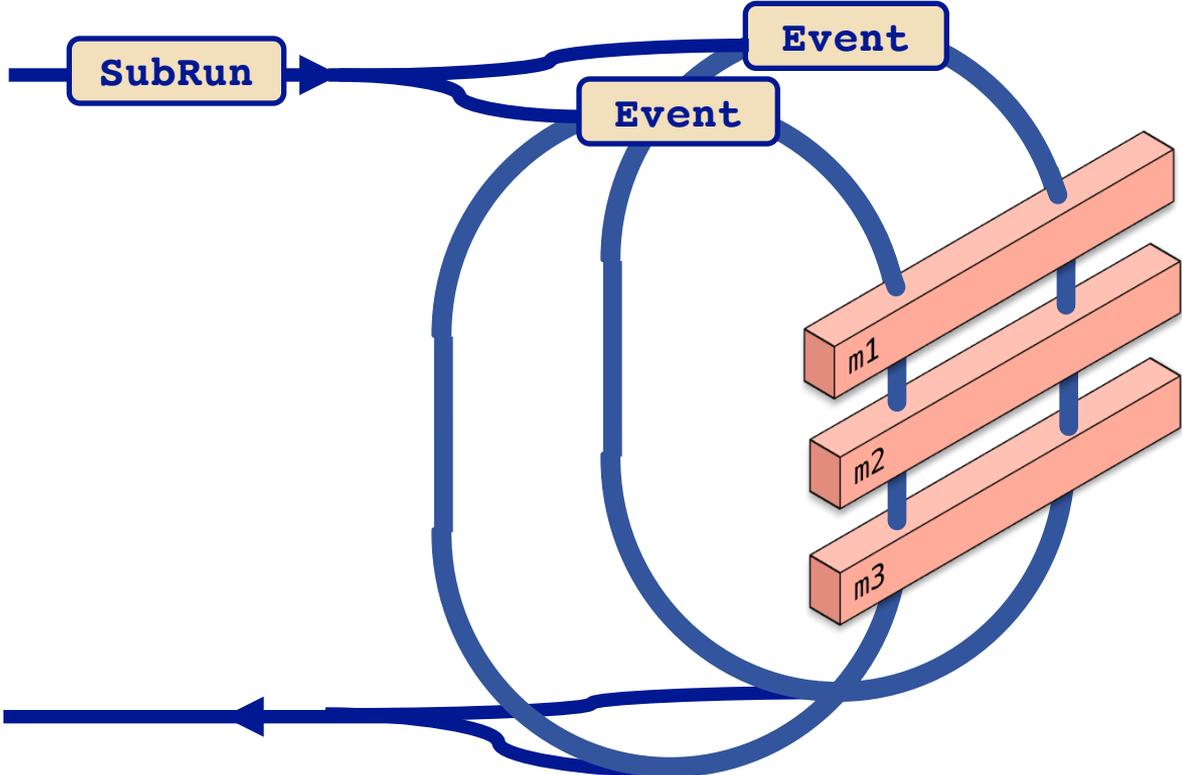
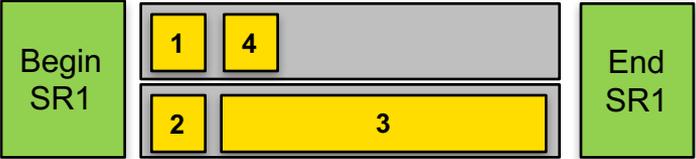
Time structure for calling modules

Multiple schedules (*art 3.0*)



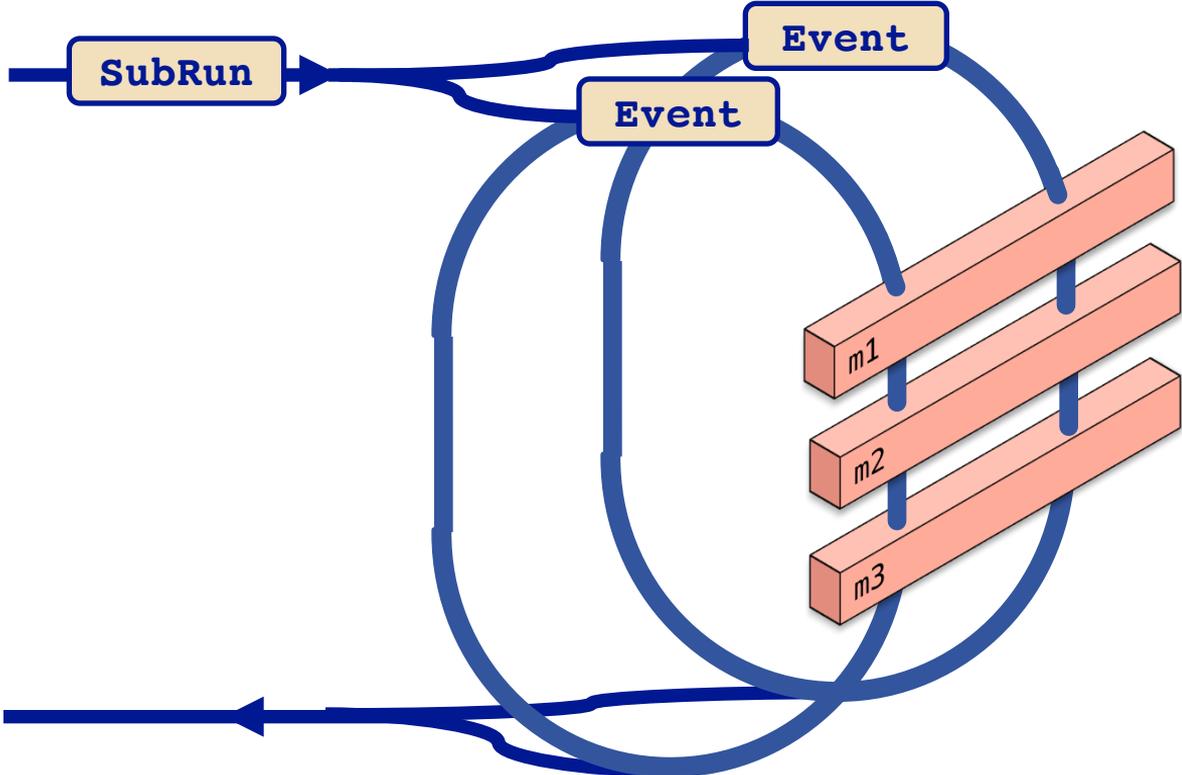
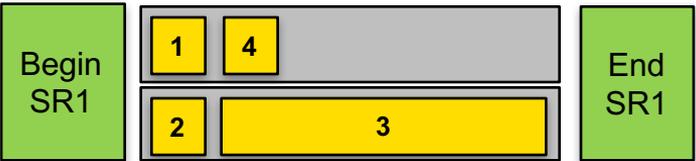
Time structure for calling modules

Multiple schedules (*art 3.0*)



Time structure for calling modules

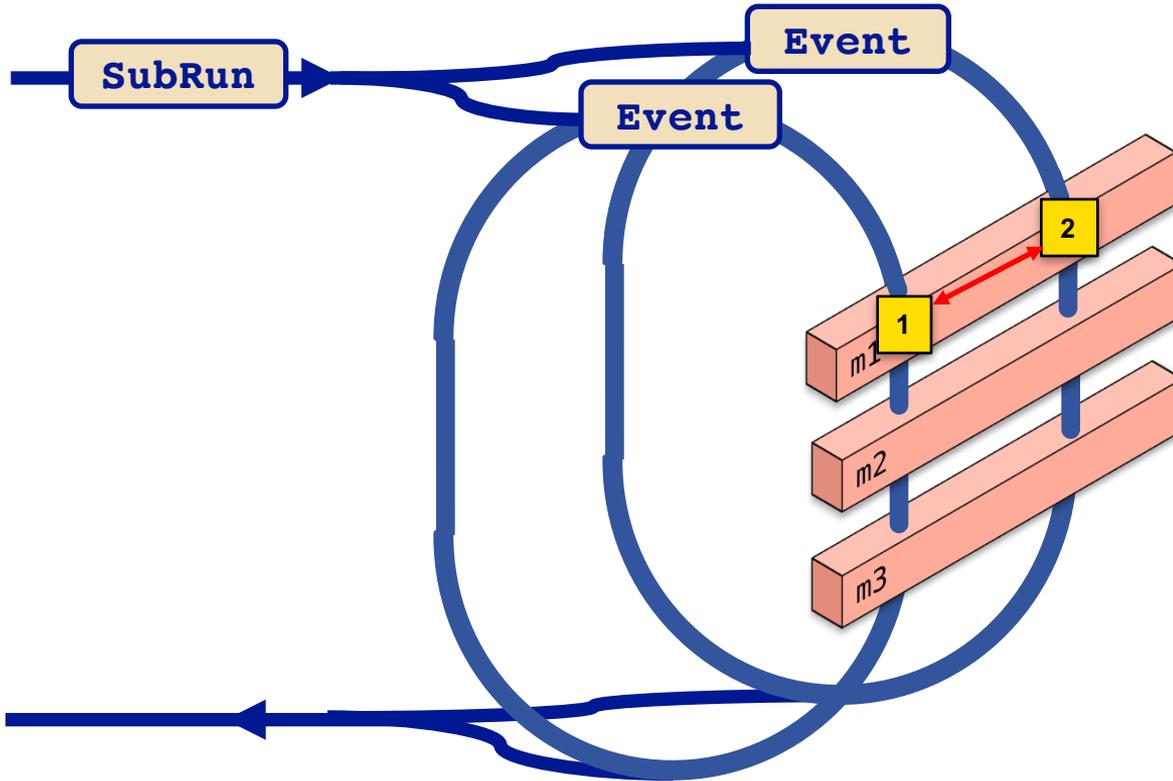
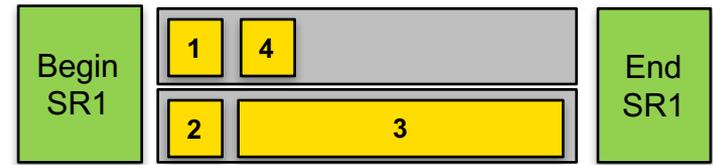
Multiple schedules (art 3.0)



Data races are now possible.

Time structure for calling modules

Multiple schedules (*art 3.0*)



If the state of one of the modules is updated when simultaneously processing two events, there can be a data race.

What are some ways to handle this?

Use a “legacy” producer

```
class HistMaker : public art::EDProducer {
public:
    explicit HistMaker(Parameters const& p)
    {}

    void produce(Event& e) override {} // Called serially
};
```

Use a “legacy” producer

```
class HistMaker : public art::EDProducer {
public:
    explicit HistMaker(Parameters const& p)
    {}

    void produce(Event& e) override {} // Called serially
};
```

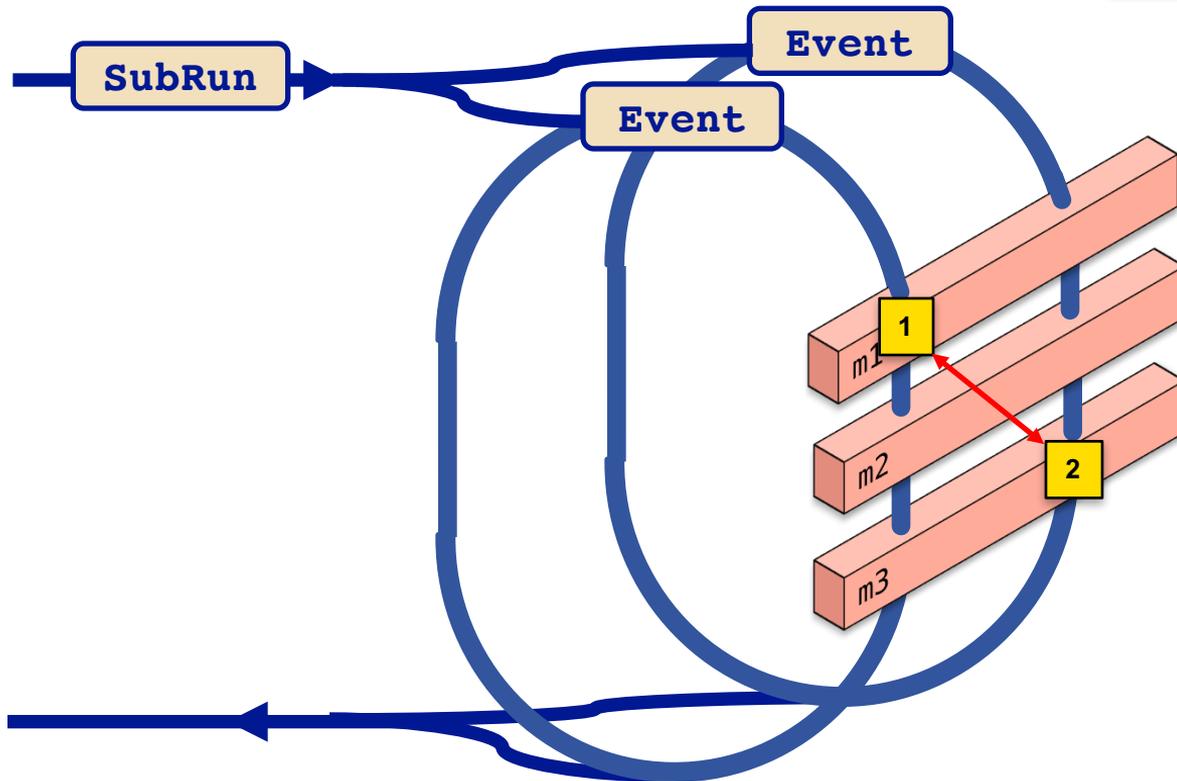
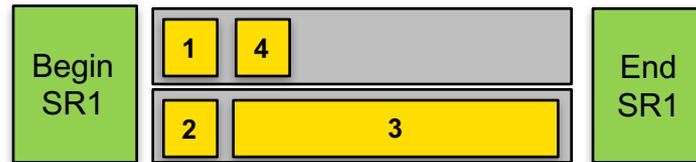
Is synonymous with...

```
class HistMaker : public art::shared::Producer {
public:
    explicit HistMaker(Parameters const& p)
    {}

    void produce(Event& e) override {} // Called serially
};
```

Time structure for calling modules

Multiple schedules (*art 3.0*)



If two modules are processing different events at the same time, but they are using a common resource, there can be a data race.

How do we avoid such a data race?

Serialized module due to shared resource

```
class Fitter : public art::shared::Producer {  
public:  
    explicit Fitter(Parameters const& p)
```

Suppose you want to call `TCollection::(Set|Get)CurrentCollection`
First step: please don't. This is only illustrating a thread-unsafe interface.

```
    // Called serially wrt. other modules that use TCollection  
    void produce(Event& e) override {}  
};
```

Serialized module due to shared resource

```
class Fitter : public art::shared::Producer {
public:
    explicit Fitter(Parameters const& p)
    {
        serialize<Event>("TCollection"); // Declare the common resource
    }

    // Called serially wrt. other modules that use TCollection
    void produce(Event& e) override {}
};
```

Serialized module due to shared resource

```
class Fitter : public art::shared::Producer {
public:
    explicit Fitter(Parameters const& p)
    {
        serialize<Event>("TCollection"); // Declare the common resource
    }

    // Called serially wrt. other modules that use TCollection
    void produce(Event& e) override {}
};
```

If you can guarantee no data races...

```
class HitMaker : public art::shared::Producer {
public:
    explicit HitMaker(Parameters const& p)
    {
        async<Event>();
    }

    void produce(Event& e) override {} // Called asynchronously
};
```

This is how you opt in to parallelism.

What things could look like eventually

```
class POTSumPlotter : public art::shared::Producer {
public:
    explicit POTSumPlotter(Parameters const& p)
    {
        serialize<SubRun>("TFileService");
        async<Event>();
    }

    void endSubRun(SubRun const& sr) override {} // Called serially
    void produce(Event& e) override {} // Called asynchronously
};
```

Replicated modules

One module per schedule

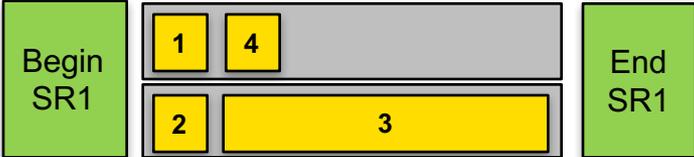
Replicated modules

One module per schedule

- Sometimes the easiest way to gain multi-threading benefits is to replicate modules across schedules—avoids data races from sharing a module.

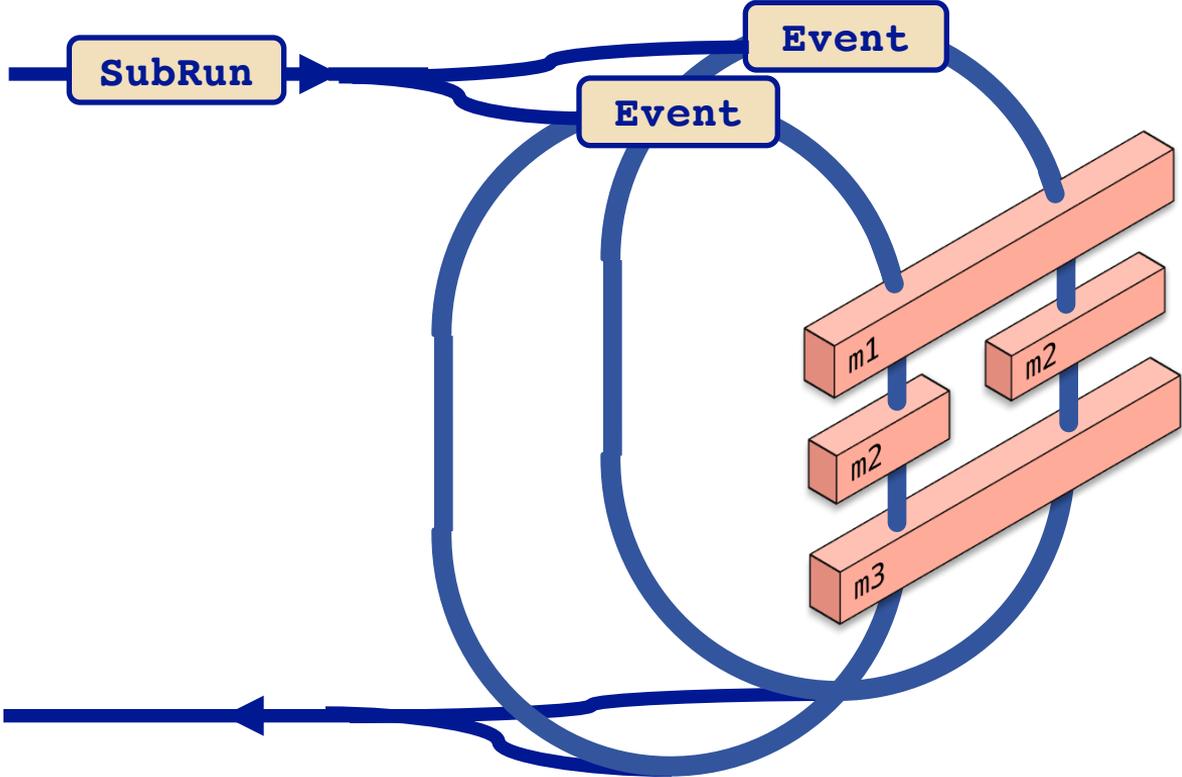
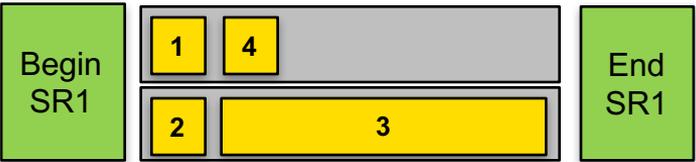
Time structure for calling modules

Multiple schedules (*art 3.0*)



Time structure for calling modules

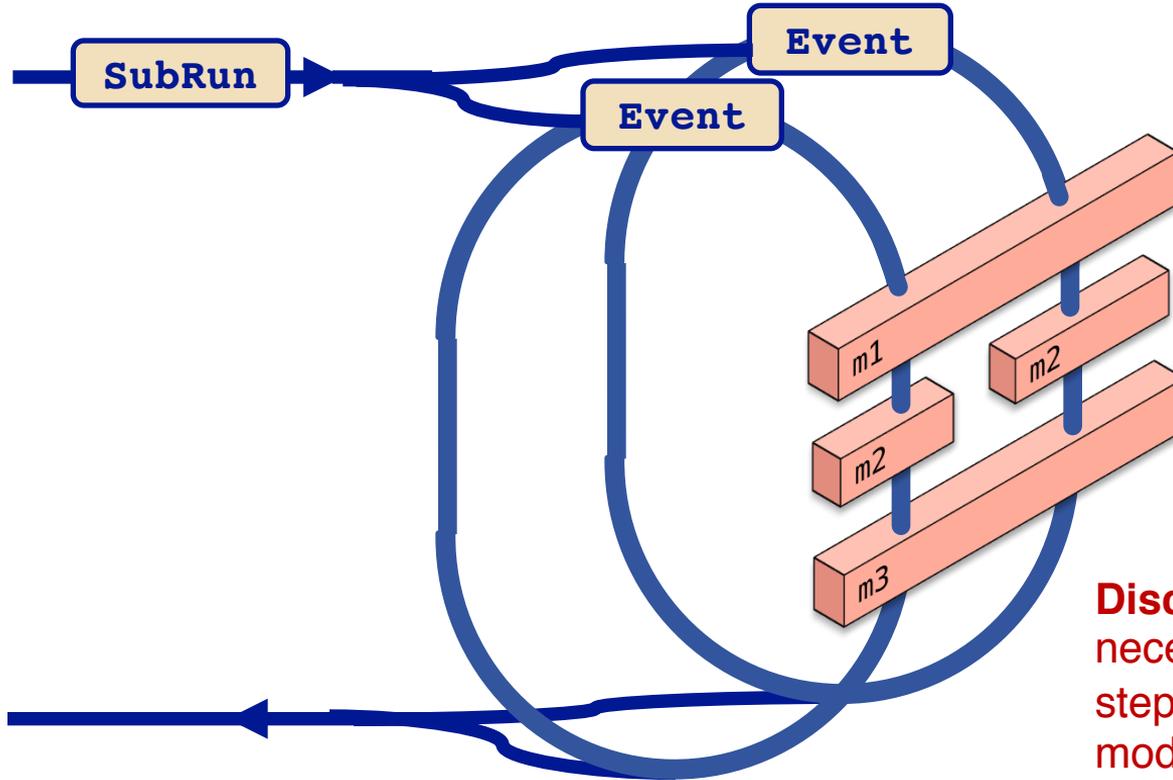
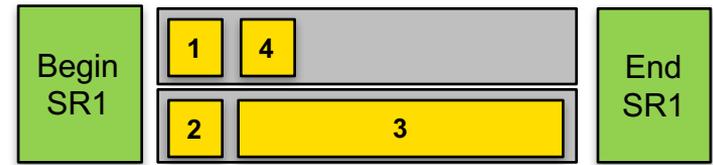
Multiple schedules (*art 3.0*)



Multiple copies of configured module **m2** avoids data-races wrt. **m2** data members.

Time structure for calling modules

Multiple schedules (*art 3.0*)



Multiple copies of configured module **m2** avoids data-races wrt. **m2** data members.

Consequence: each module copy does not see all events.

Discussion topic: It may be necessary to provide a *reduction* step where data members from the module copies can be combined.

Replicated producer

```
class Accumulator : public art::replicated::Producer {
public:
    explicit Accumulator(Parameters const& p)
    {}

    // Each module copy sees one event at a time
    void produce(Event& e) override;

    // Reduction interface to be discussed...
};
```

Some closing remarks

- We have taken care to design a system that:
 - Is based off of previous CMSSW experience
 - Is as simple as possible to “opt in” to and understand
 - Can still be very flexible
- We are in the process of converting our integration tests to use multi-threaded execution, testing the system.
- There are still issues to be resolved, but the primary infrastructure for MT *art* is in place.
- With *art* 2.10, we intend to implement the MT-based task-scheduling, without introducing any multi-threading interface or behavior to the user.

Future steps

- We are considering allowing the insertion of data products via services during the post-read callbacks.
 - These callbacks are invoked in a protected manner—*i.e.* no data races
 - Implementing can facilitate easier interaction with conditions information.
- Before the next meeting, we will be sending out the questions we want to discuss and answer:
 - Is a reduction facility needed for replicated modules?
 - Are there specific constraints your experiment has regarding processing sub-runs concurrently?
 - What is the desired reproducibility/replayability behaviors for the `RandomNumberGenerator` service?
 - Do you explicitly use `TriggerResults` objects?