Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

Multi-threaded *art*

Kyle J. Knoepfel

25 June 2019

LArSoft Workshop 2019

Outline

- *art*'s path processing
  - Consequences

- *art*'s multi-threading behavior
  - Command-line invocation
  - Guarantees and limitations
  - Kinds of modules
    - Illustrations
  - Services

- Guidance moving to multi-threaded *art* programs

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: {...}
  }

  hitPath: [makeHits, makeShowers]
  geomPath: [produceG4Steps]
  analyzePath: [plotHits]
}
```

Trigger path
Trigger path
End path

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]
}
```

Trigger path
Trigger path
End path

- **The order in which *trigger paths* are executed is unspecified (single-threaded).**
- **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.
- A module is executed once per event.

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]
```

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

- **The order in which *trigger paths* are executed is unspecified (single-threaded).**
- **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.
- A module is executed once per

Heeding these facts is essential for successful use of *art* 3.

Consequences of *art*'s guarantees

- Modules on one trigger path may not consume products created by modules that are not on that same path.

Consequences of *art*'s guarantees

- Modules on one trigger path may not consume products created by modules that are not on that same path.
- The following is a configuration error (heuristically):

```
physics: {
  producers: {
    p1: { produces: ["int", ""] }
    p2: { consumes: ["int", "p1::current_process"] }
  }
  tp1: [p1]
  tp2: [p2]
}
```

Consequences of *art*'s guarantees

- Modules on one trigger path may not consume products created by modules that are not on that same path.
- The following is also a configuration error (heuristically):

```
physics: {
  producers: {
    p1: { produces: ["int", ""] }
    p2: { produces: ["int", "instanceName"] }
    readThenMake: {
      consumesMany: ["int"] // calls getMany
    }
  }
  tp1: [p1, readThenMake]
  tp2: [p2, readThenMake]
}
```

Consequences of *art*'s guarantees

- Modules on one trigger path may not consume products created by modules that are not on that same path.
- The following is also a configuration error (heuristically):

```
physics: {
    }
  }
  tp1: [p1, readThenMake]
  tp2: [p2, readThenMake]
}
```

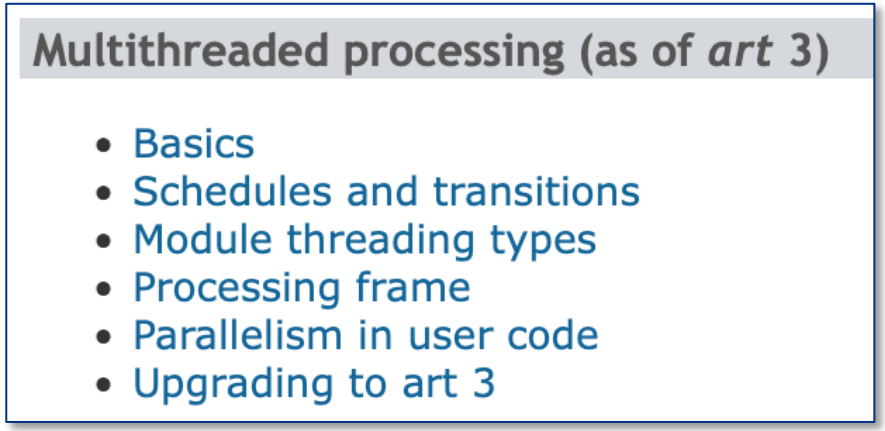
art 3 catches these errors if you use the `consumes` interface.

```
Module readThenMake on paths tp1, tp2 depends on
   Module p2 on path tp2
```

art's multi-threading behavior

art's multi-threading behavior

https://cdcvs.fnal.gov/redmine/projects/art/wiki#Multithreaded-processing-as-of-art-3

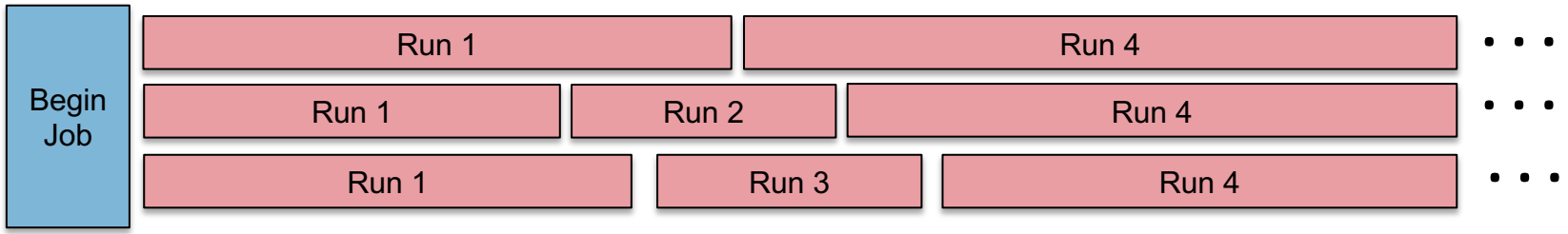


Fermilab

The design

- 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 event loops (*schedule*s) and (optionally) the maximum number of threads that the process can use.
- **Each schedule processes one event at a time.**

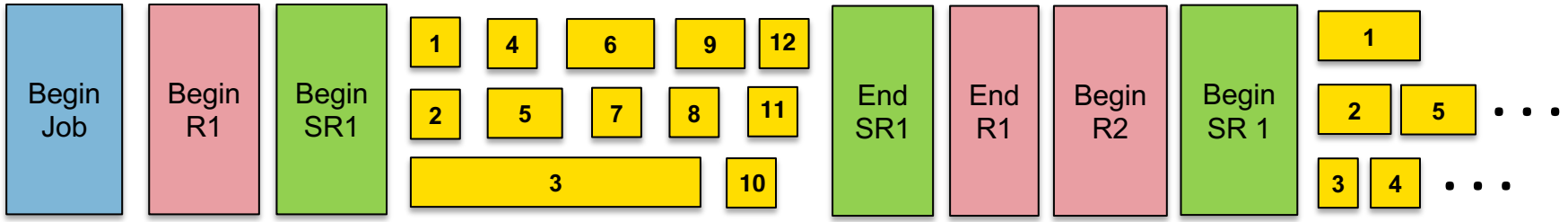
Our goal:



The design

- 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 event loops (*schedule*s) and (optionally) the maximum number of threads that the process can use.
- **Each schedule processes one event at a time.**

Currently implemented:

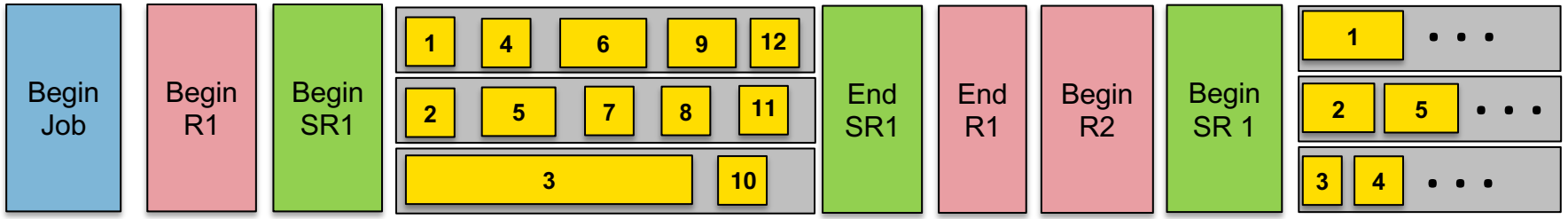


Fermilab

The design

- 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 event loops (*schedule*s) and (optionally) the maximum number of threads that the process can use.
- **Each schedule processes one event at a time.**

Currently implemented:



Fermilab

The design

- 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 event loops (*schedule*s) and (optionally) the maximum number of threads that the process can use.
- **Each schedule processes one event at a time.**
- Different modules can be run in parallel on the *same* event.
- Users are allowed to use TBB's parallel facilities within their own modules.

Multi-threaded event-processing

- *art* 3 supports concurrent processing of events.
  - The number of events to process concurrently is specified by the **number of schedules**
  - The user can optionally specify the number of threads.
- The user ***opts in*** to concurrent processing.

Multi-threaded event-processing

- *art* 3 supports concurrent processing of events.
  - The number of events to process concurrently is specified by the **number of schedules**
  - The user can optionally specify the number of threads.
- The user ***opts in*** to concurrent processing.

Command	(nSch, nThr)
`art -c <config> …`	(1, 1)
`art -c <config> -j 1 …`	(1, 1)
`art -c <config> -j 4 …`	(4, 4)
`art -c <config> -j 0 …`	(nproc, nproc)
`art -c <config> --nschedules 1 --nthreads 4 …`	(1, 4)

- In a grid environment, number of threads is limited to the number of CPUs configured for the HTCondor slot (*art* adjusts the number of threads).

art 3 guarantees

- Processing of an event happens on one and only one schedule.
- For a given trigger path, modules are processed in the order specified.
- A module shared among paths will be processed only once per event.
- Product insertion into the event is thread-safe.
- Product retrieval from the event is thread-safe.
- Provenance retrieval from the event is thread-safe.
- All modules and services provided by *art* are thread-safe.
  - For `TFileService`, the user is required to specify additional serialization.

art 3 limitations—*Primum non nocere (first, to do no harm)*

- Only events within the same SubRun are processed concurrently.
- Analyzers and output modules do not run concurrently.

- Other details
  - `MixFilter` modules are legacy modules.
  - Secondary input-file reading is allowed only for 1 schedule and 1 thread.
  - `TFileService` file-switching is allowed only for 1 schedule and 1 thread.

Kinds of modules in *art* 3

- *art* guarantees that any currently-existing modules are usable in a multi-threaded execution of art.
  - No multi-threading benefits are realized with legacy modules

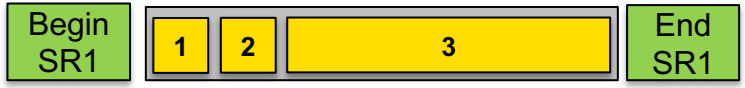
- To take advantage of *art*'s multi-threading capabilities, users will need to choose the kind of module they use:

  - **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.

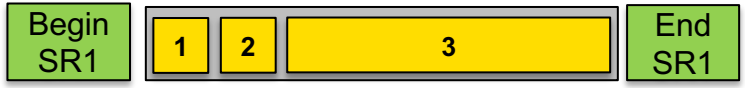
Time structure for calling modules

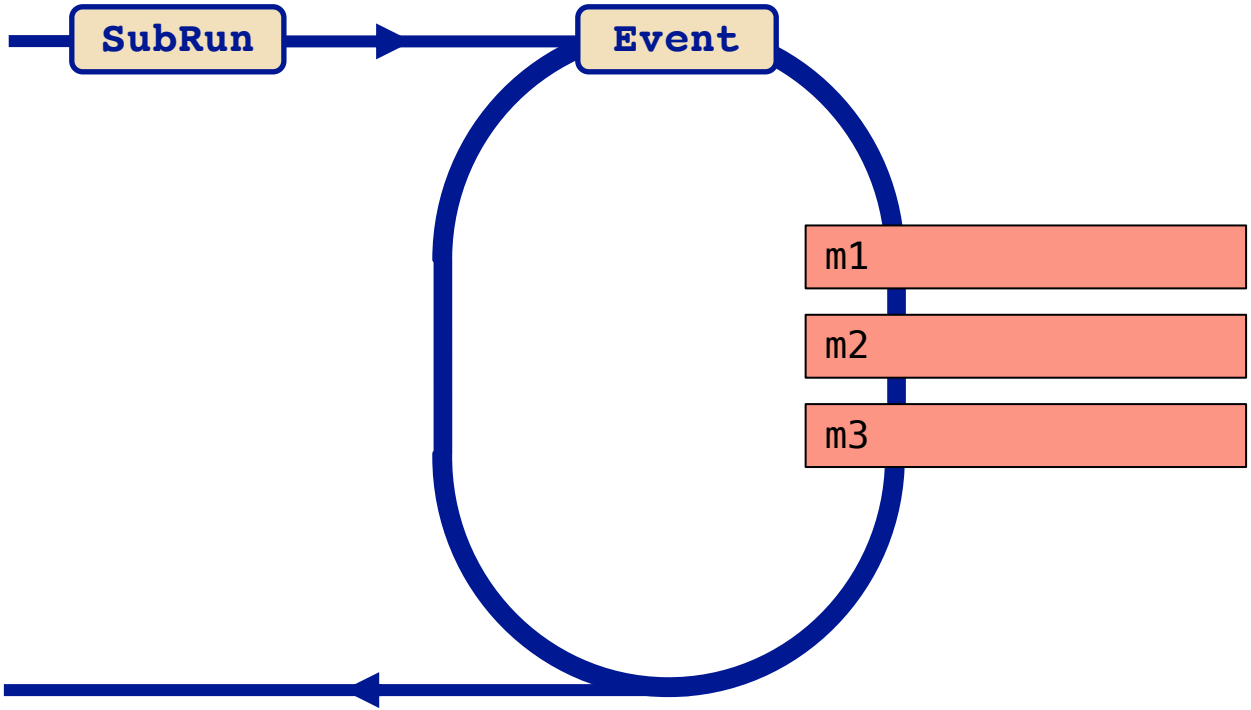
Single schedule



Time structure for calling modules

Single schedule





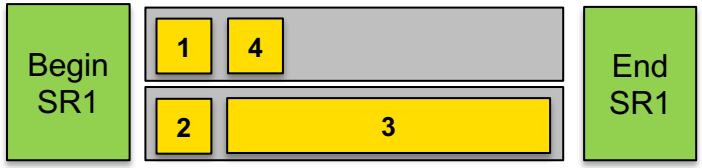
Fermilab

Shared modules

Modules shared across schedules

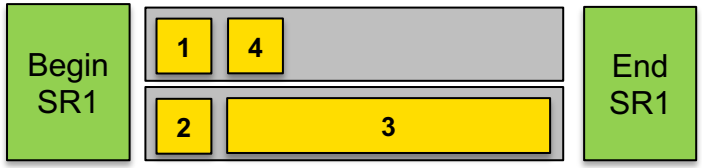
Time structure for calling modules

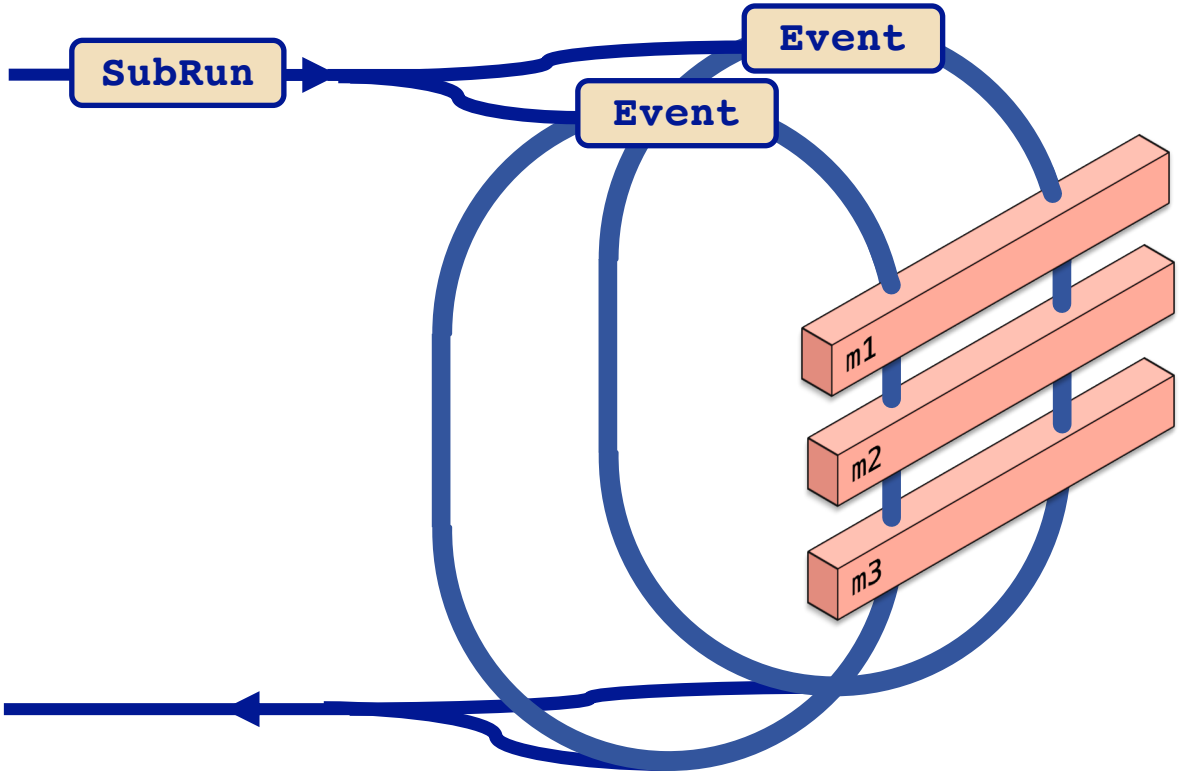
Multiple schedules



Time structure for calling modules

Multiple schedules

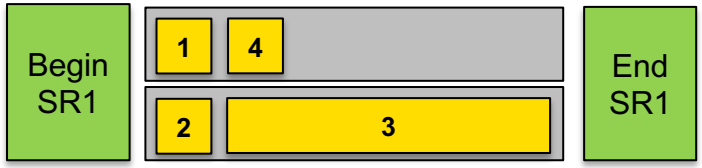


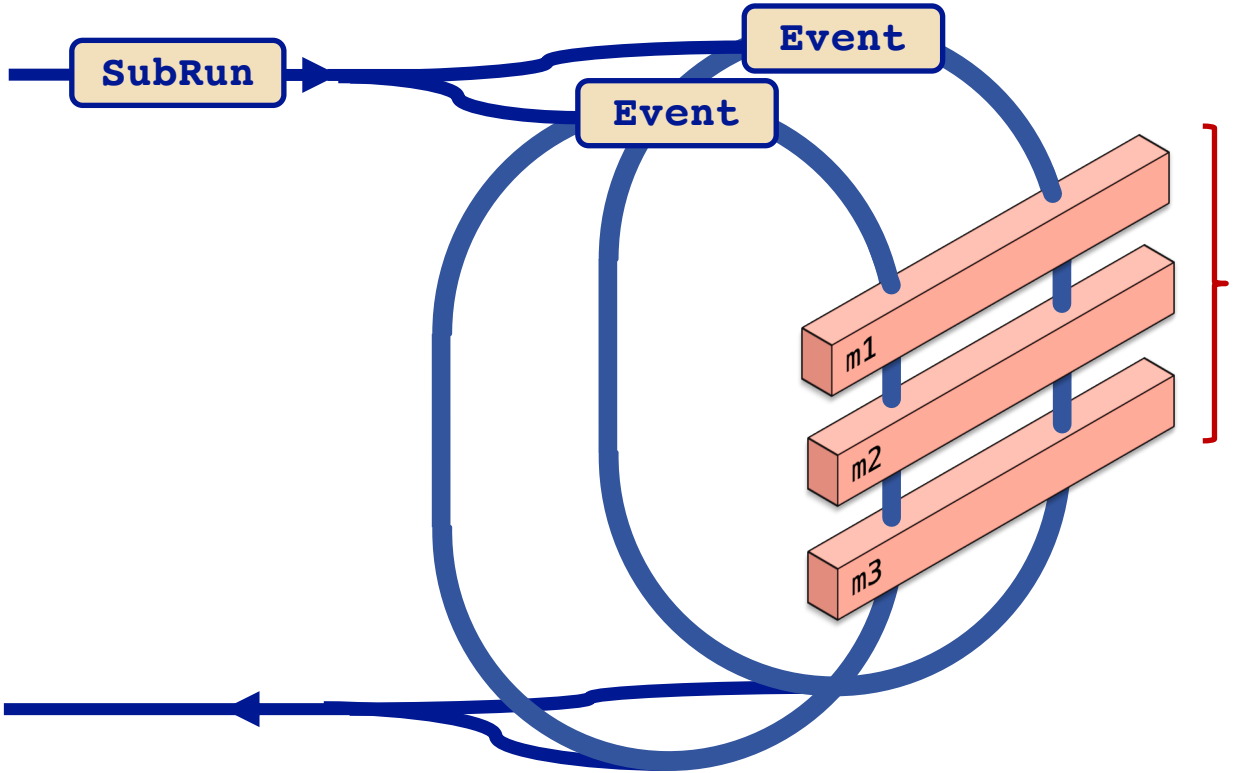


Fermilab

Time structure for calling modules

Multiple schedules



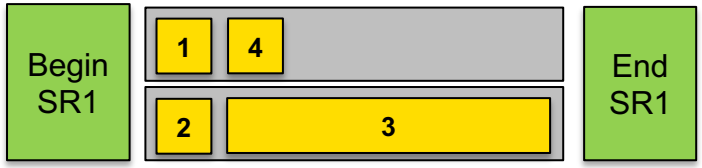


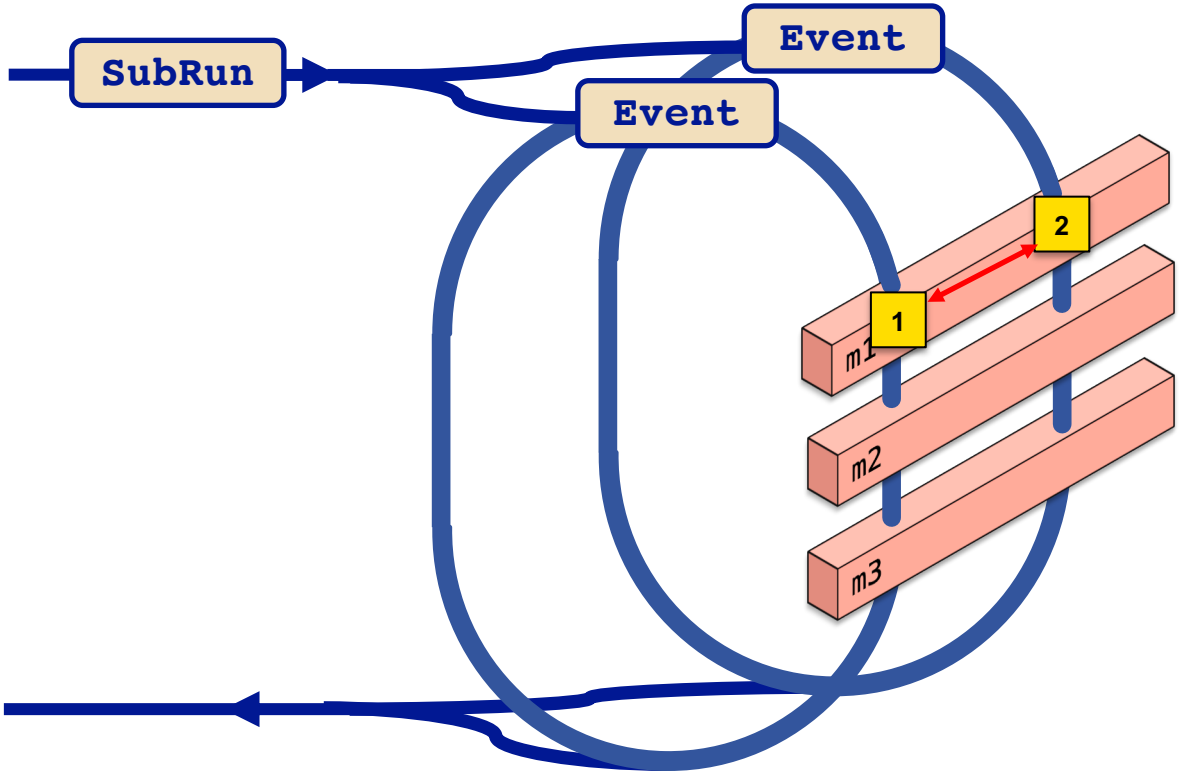
Data races are now possible.

Fermilab

Time structure for calling modules

Multiple schedules





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?

Fermilab

Using a legacy module

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

  void produce(Event& e) override {} // Called serially wrt. all
                                     // serialized modules
};
```

- Legacy modules imply maximum serialization.
  - Legacy modules cannot be run in parallel with any other legacy modules or any serialized shared modules.
- With *art* 3, any new modules should not be legacy modules.
- The better solution is to use a `SharedModule`, which can be serialized only wrt itself.

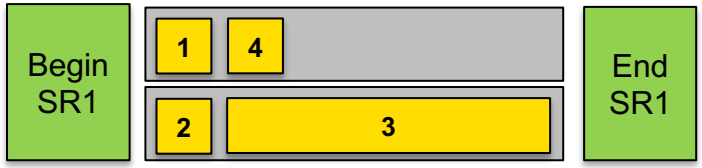
Use a shared module

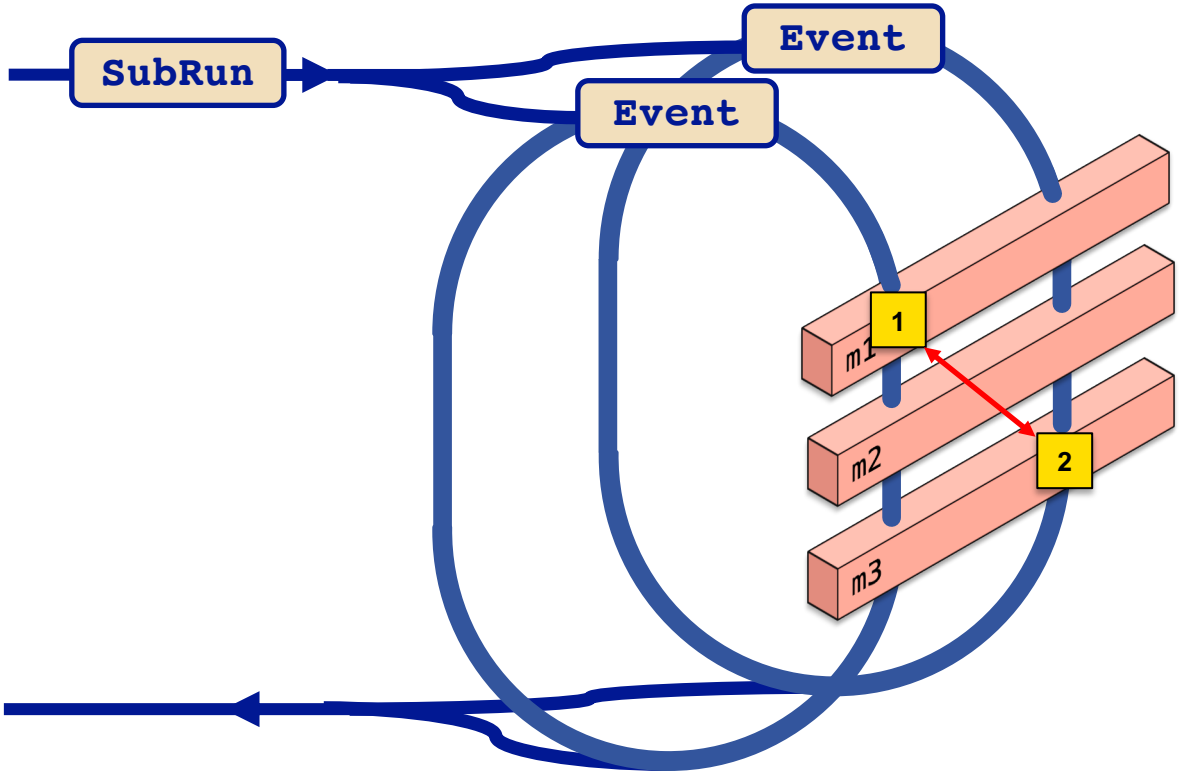
```
class HistMaker : public art::SharedProducer {
public:
  explicit HistMaker(Parameters const& p,
                     ProcessingFrame const&) : SharedProducer{p}
  {
    serialize<InEvent>(); // Declaration to process
                          // one event at a time.
  }

  // Called serially wrt. itself
  void produce(Event&, ProcessingFrame const&) override;
};
```

- But there can be other data race problems.

Time structure for calling modules

Multiple schedules





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?

Fermilab

Serialized module due to shared resource

Serialized module due to shared resource

```
class Fitter : public art::SharedProducer {
public:
  explicit Fitter(Parameters const& p,
                  ProcessingFrame const& frame) : SharedProducer{p
    consumes<TCollection>(); // declare the common resource
  }

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

Suppose you want to call **`TCollection::(Set|Get)CurrentCollection`**

First step: please don't. This is only illustrating a thread-unsafe interface.

Serialized module due to shared resource

```
class Fitter : public art::SharedProducer {
public:
  explicit Fitter(Parameters const& p,
                  ProcessingFrame const& frame) : SharedProducer{p}
  {
    serialize<InEvent>("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::SharedProducer {
public:
  explicit HitMaker(Parameters const& p ,
                    ProcessingFrame const&) : SharedProducer{p}
  {
    async<InEvent>();
  }

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

Replicated modules

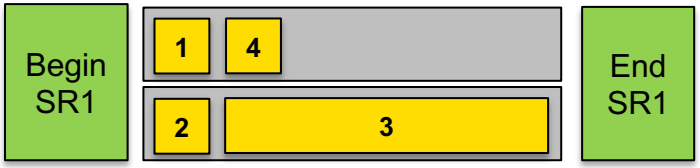
One module per schedule

Replicated modules

One module per schedule

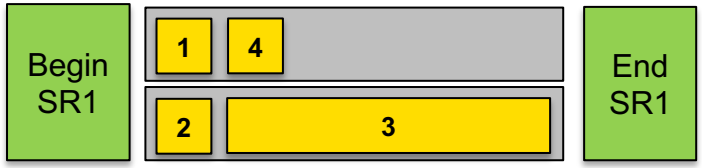
- Sometimes the easiest way to gain multi-threading benefits is to replicate modules across schedules—avoids data races from sharing a module.

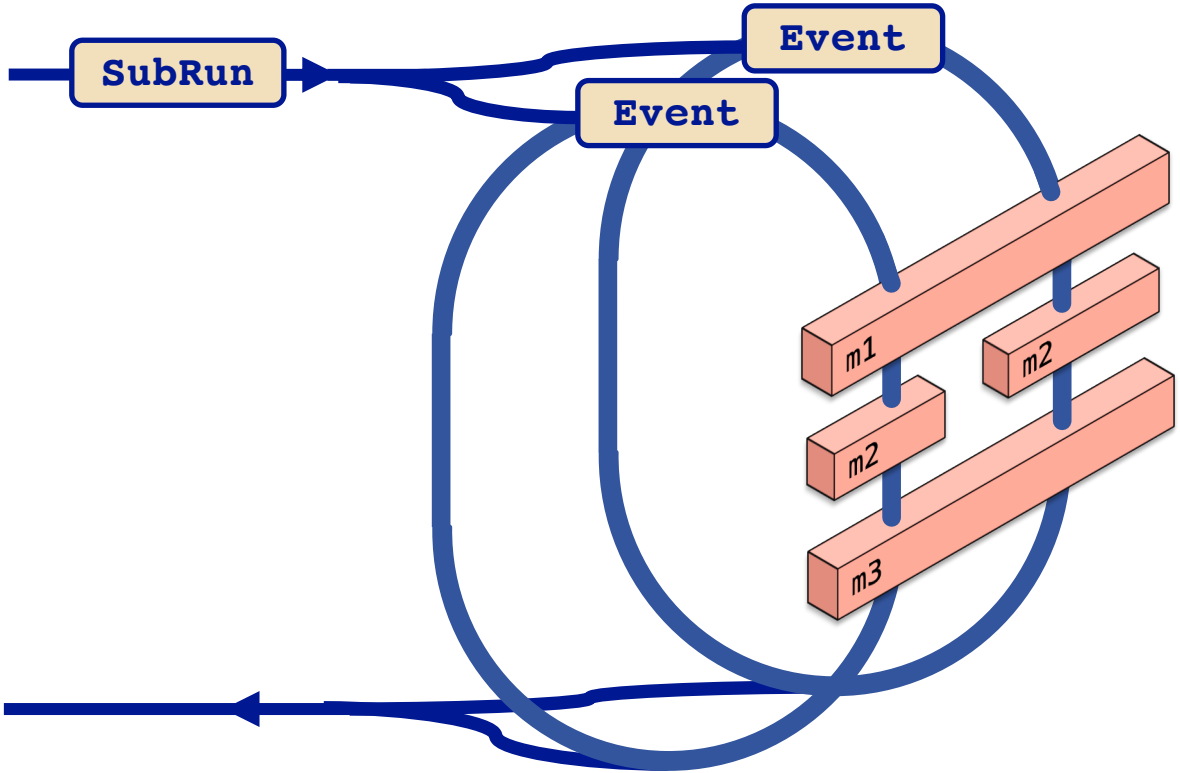
Time structure for calling modules
Multiple schedules



Time structure for calling modules

Multiple schedules



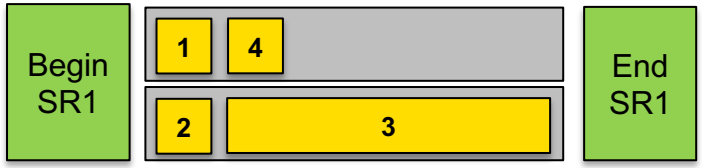


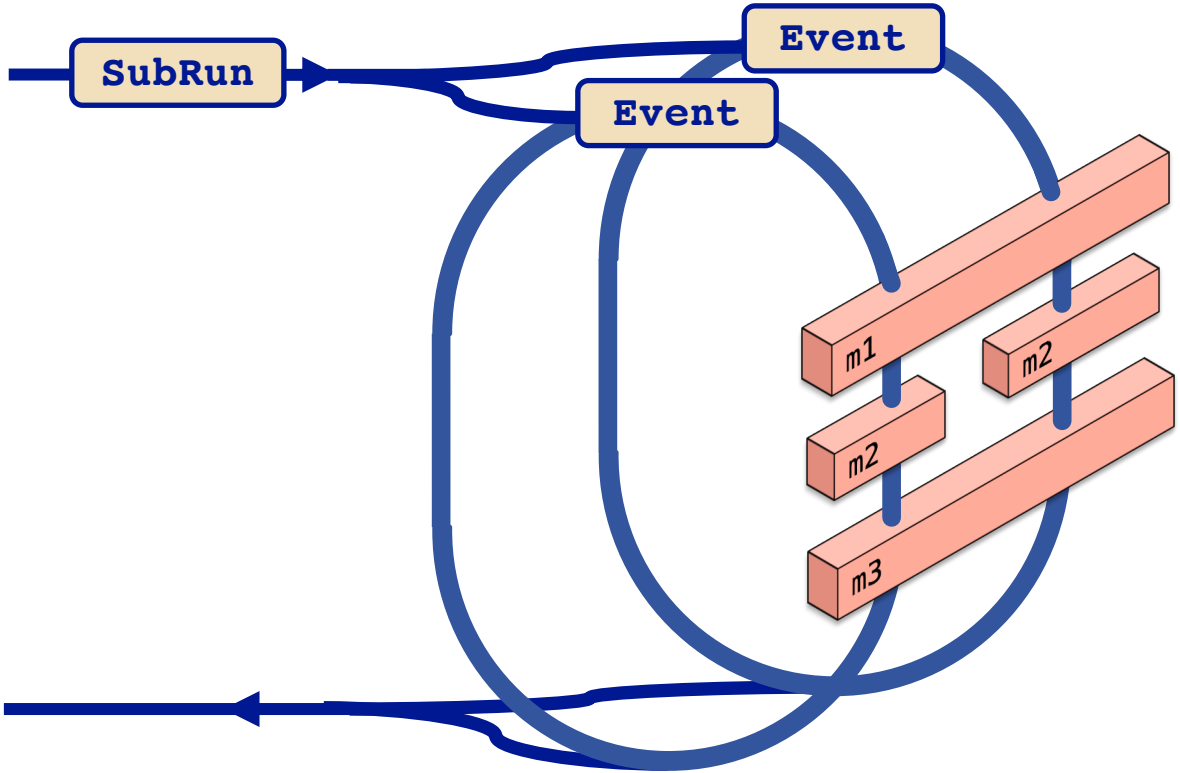
Multiple copies of configured module **`m2`** avoids data-races wrt. **`m2`** data members.

Fermilab

Time structure for calling modules

Multiple schedules





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

Consequence: each module copy does not see all events.

Fermilab

Replicated producer

```
class Accumulator : public art::ReplicatedProducer {
public:
  explicit Accumulator(Parameters const& p,
                       ProcessingFrame const& frame)
    : ReplicatedProducer{p, frame}
  {}

  // Each module copy sees one event at a time
  void produce(Event&, ProcessingFrame const&) override;
};
```

- Do not use a replicated producer is you need to use a shared resource.
- For *art* 3.0, replicated modules cannot produce Run and SubRun data products.

What is the `ProcessingFrame` type?

> *"O `art::ServiceHandle<T>{}`, thou time is short."*
> *- Anonymous*

- Until now, users have been able to create `ServiceHandles` from anywhere; this pattern is changing.
- The recommended pattern is for *art* users to create service handles from the passed-in `ProcessingFrame` object.

```
void HitMaker::beginRun(Run&, ProcessingFrame const& frame)
{
  auto h1 = frame.serviceHandle<Calib>();       // => ServiceHandle<Calib>
  auto h2 = frame.serviceHandle<Calib const>(); // => ServiceHandle<Calib const>
}
```

- This will eventually allow for replicated services, akin to replicated modules.

Services

- Services are globally shared objects (across schedules and threads).
  - They can be accessed from anywhere through a ServiceHandle.
  - They must be thread-safe.

Services

- Services are globally shared objects (across schedules and threads).
  - They can be accessed from anywhere through a ServiceHandle.
  - They must be thread-safe.

*LArSoft's prevalent use of **mutable** services is the primary limitation in realizing multi-threading benefits.*

Services

- Services are globally shared objects (across schedules and threads).
  - They can be accessed from anywhere through a ServiceHandle.
  - They must be thread-safe.

*LArSoft's prevalent use of **mutable** services is the primary limitation in realizing multi-threading benefits.*

- In order to use a service in an art job, with more than one schedule/thread enabled, the service must be GLOBAL (SHARED, for art 3.03).
- LEGACY services are supported only in single-schedule/single-threaded mode.

```
---- Configuration BEGIN
  The service 'MyService' is a legacy service,
  which can be used with only one schedule and one thread.
  This job uses 2 schedules and 2 threads.
  Please reconfigure your job to use only one schedule/thread.
---- Configuration END
```

ROOT and MT

- ROOT's thread-safety flag **has** been enabled by *art*.
  - Allows (e.g.) multiple ROOT files to be opened in parallel.
- ROOT's implicit MT flag **has not** been enabled by *art.*
- All interactions *art* has with ROOT are serialized.
  - Input-file reading
  - Output-file writing
  - To use `TFileService`, you must use a shared module that calls the appropriate `serialize` function.

Guidance moving to *art* 3

- **Solve workflow issues first.**
  - You might have thread-safe modules and services.
  - If you're relying on illegal path configurations, you'll run into product dependency errors.

Guidance moving to *art* 3

- **Solve workflow issues first.**
  - You might have thread-safe modules and services.
  - If you're relying on illegal path configurations, you'll run into product dependency errors.

Recompile/rerun jobs with 1 schedule/1 thread (default)

Add consumes statements to modules (use –M program option for help)

Recompile/rerun jobs with 1 schedule/1 thread and use `--errorOnMissingConsumes`

Recompile/rerun jobs with more than 1 schedule/1 thread

Guidance moving to *art* 3

- **Solve workflow issues first.**
  - You might have thread-safe modules and services.
  - If you're relying on illegal path configurations, you'll run into product dependency errors.
- **Determine what kind of module you need.**
  - Producer, filter, or analyzer?
  - Do you need to create (Sub)Run products?
  - Do you need to see every event?
  - Do you need to call an external library that is not thread-safe?
  - Do you have mutable data members for which operations are not thread-safe?

Guidance moving to *art* 3

- **Solve workflow issues first.**
  - You might have thread-safe modules and services.
  - If you're relying on illegal path configurations, you'll run into product dependency errors.
- **Determine what kind of module you need.**
  - Producer, filter, or analyzer?
  - Do you need to create (Sub)Run products?
  - Do you need to see every event?
  - Do you need to call an external library that is not thread-safe?
  - Do you have mutable data members for which operations are not thread-safe?

- We can provide guidance in dealing with such issues.
- **Contact us**.