The detector-clocks service

A case study in determining thread-safe service access patterns

Kyle J. Knoepfel

17 December 2019

LArSoft coordination meeting

Services

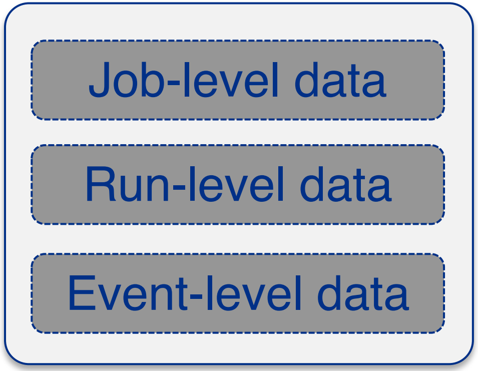
- The SciSoft team has been working toward making LArSoft code thread-safe.

- Services are problematic due to widespread use of non-const mutable data.
  - `DetectorClocks` and `DetectorProperties` suffer from this malady.

- In this talk, I will present:
  - A pattern that can be adopted for both services to make them thread-safe.
  - My work toward that end for the `DetectorClocks` service.
  - A proposal for adopting the pattern.

Thread-unsafe approach

- Monolithic data structures are often chosen for managing *mutable* data corresponding to different processing granularities.



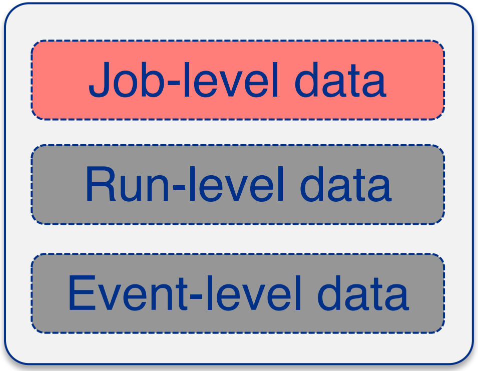
Job-level data

Run-level data

Event-level data

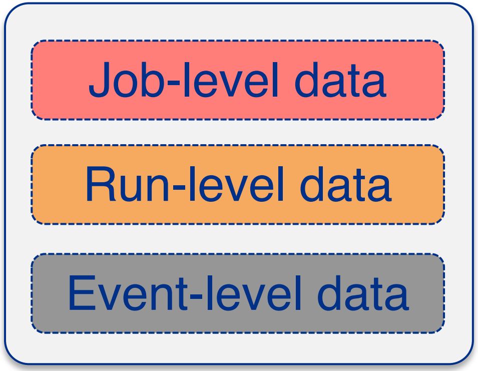
- This is true for various LArSoft facilities (e.g. `DetectorClocks` and `DetectorProperties`).
- It is inherently thread-*unsafe* as it often relies on the notion of “current”, which is ill-defined in multi-threaded environments.

Thread-unsafe approach



Thread 1	Thread 2
Create service	

Thread-unsafe approach



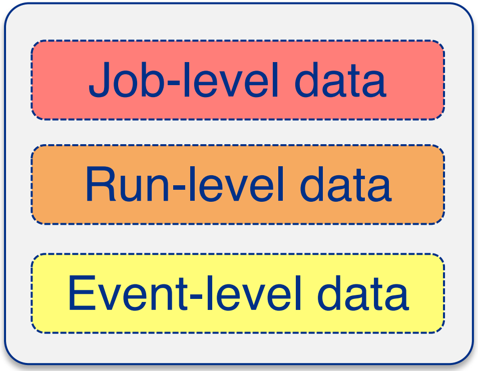
Job-level data

Run-level data

Event-level data

Thread 1	Thread 2
Begin Run 1	

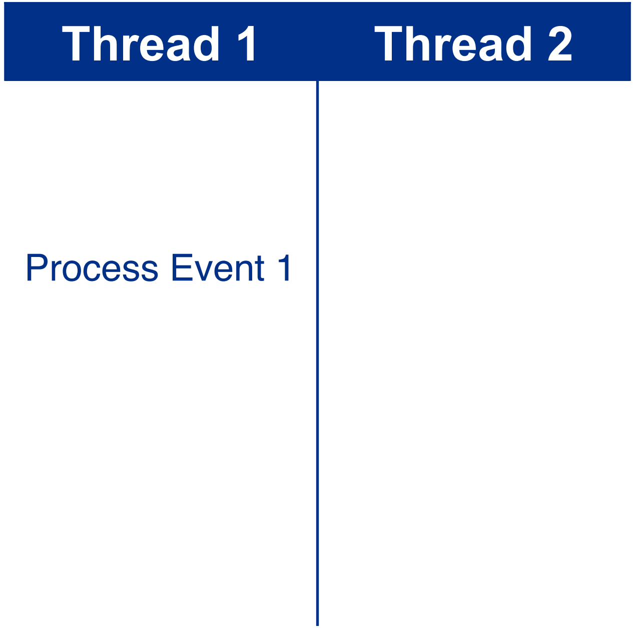
Thread-unsafe approach



Job-level data

Run-level data

Event-level data

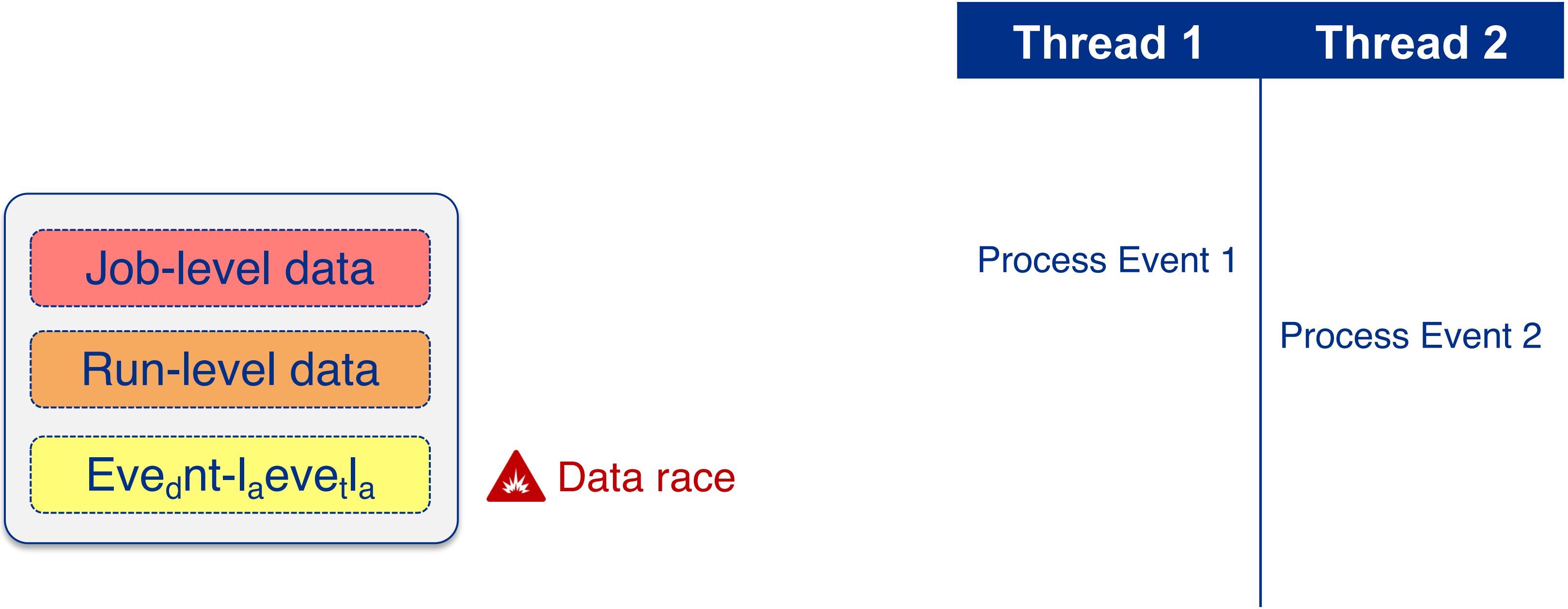


Thread 1

Thread 2

Process Event 1

Thread-unsafe approach



Thread 1

Thread 2

Job-level data

Run-level data

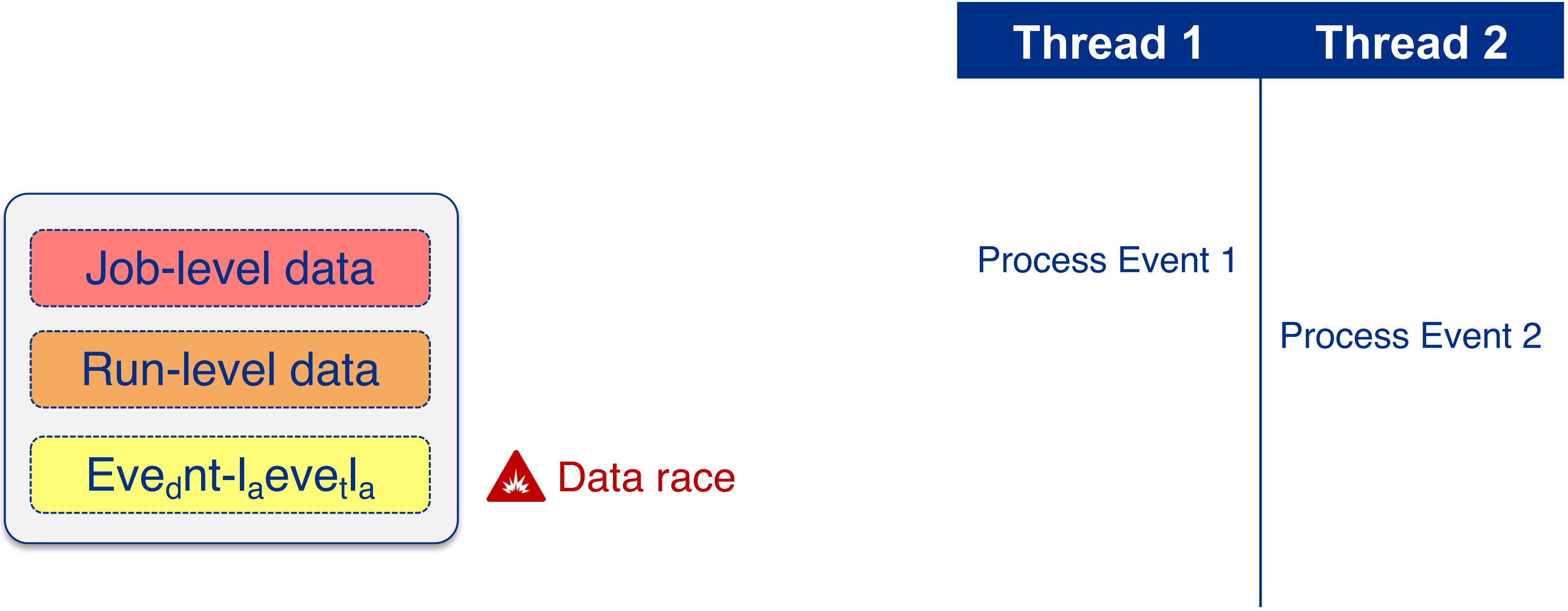
Eve$_{d}$nt-l$_{a}$eve$_{t}$l$_{a}$

Data race

Process Event 1

Process Event 2

Thread-unsafe approach



- To solve this problem for the `DetectorClocks` provider/service, I have adopted the “persistent data structure” approach.
  - Data structures broken up according to the processing steps required.
  - In what follows, all boxes represent immutable objects.

Persistent data structure approach

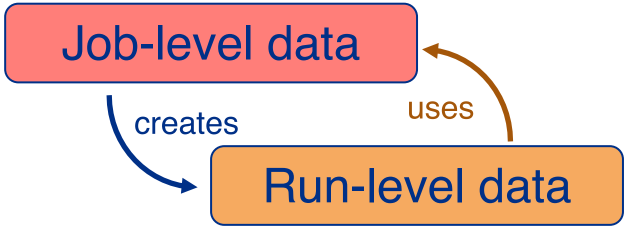
Thread 1	Thread 2

Persistent data structure approach

Job-level data

Thread 1	Thread 2
Create service	

Persistent data structure approach

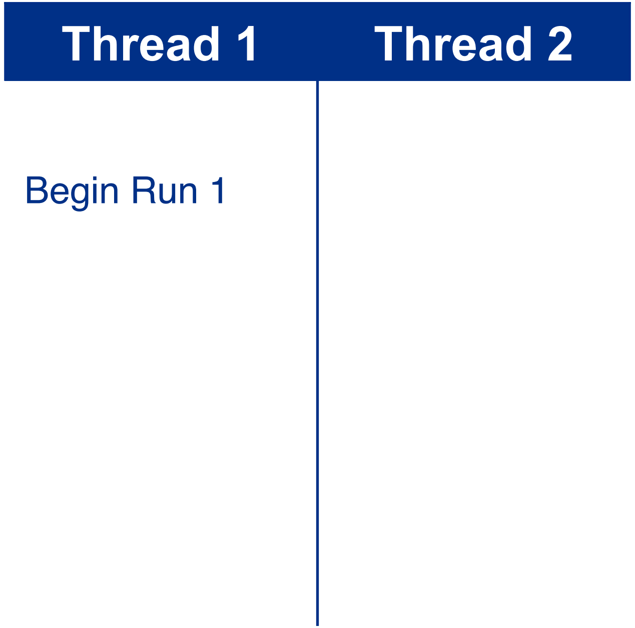


Job-level data

creates

uses

Run-level data

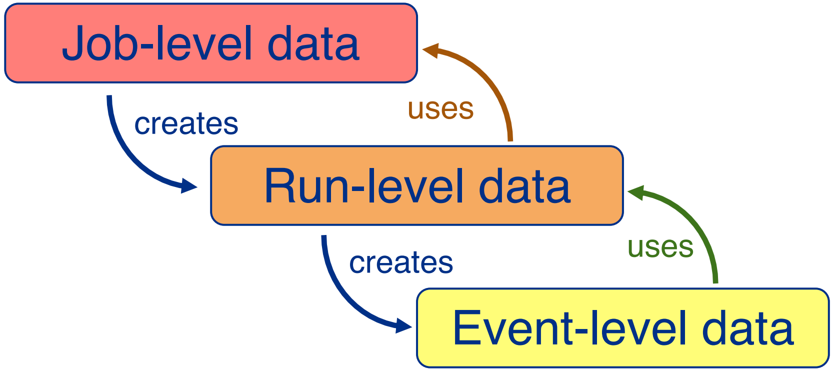


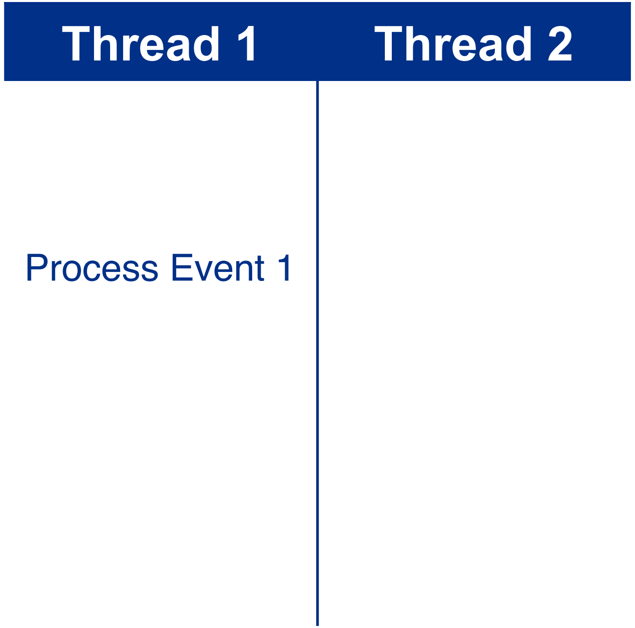
Thread 1

Thread 2

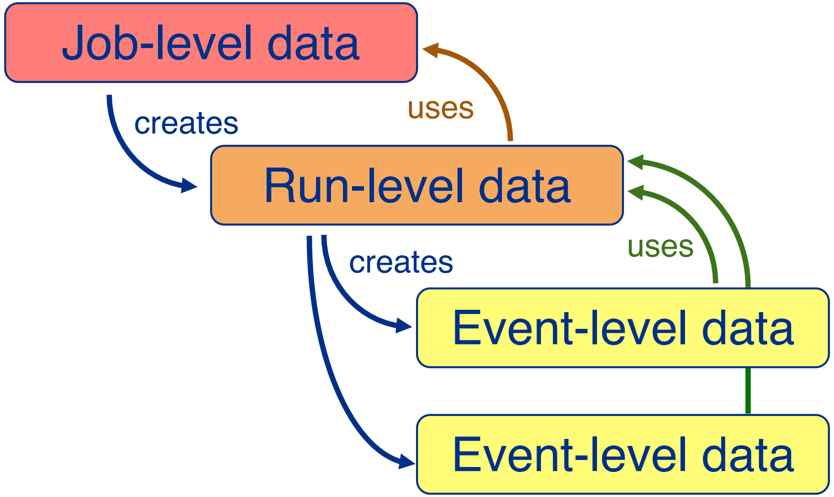
Begin Run 1

Persistent data structure approach





Persistent data structure approach



Job-level data

creates

uses

Run-level data

creates

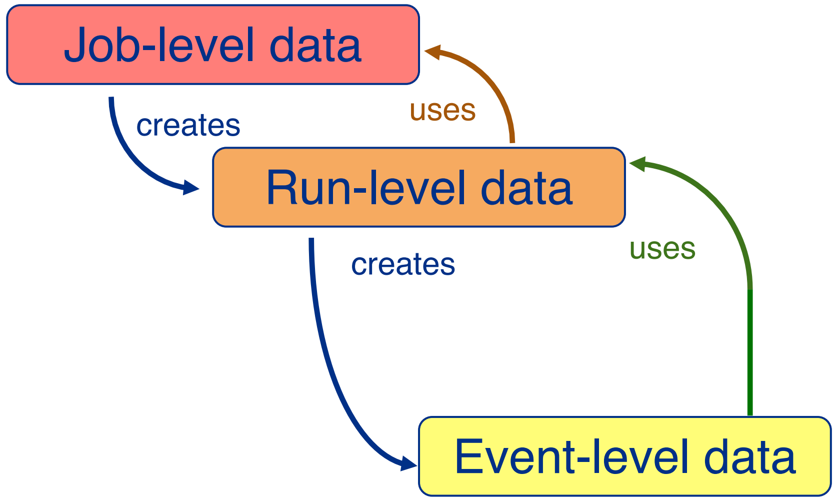
uses

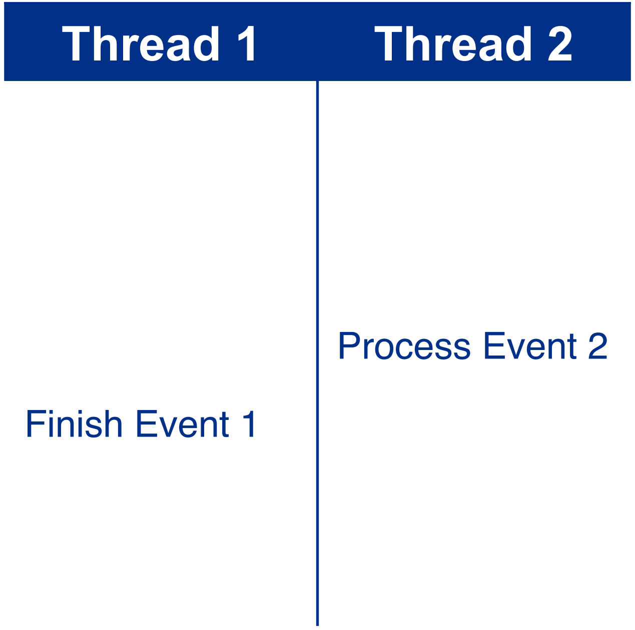
Event-level data

Event-level data

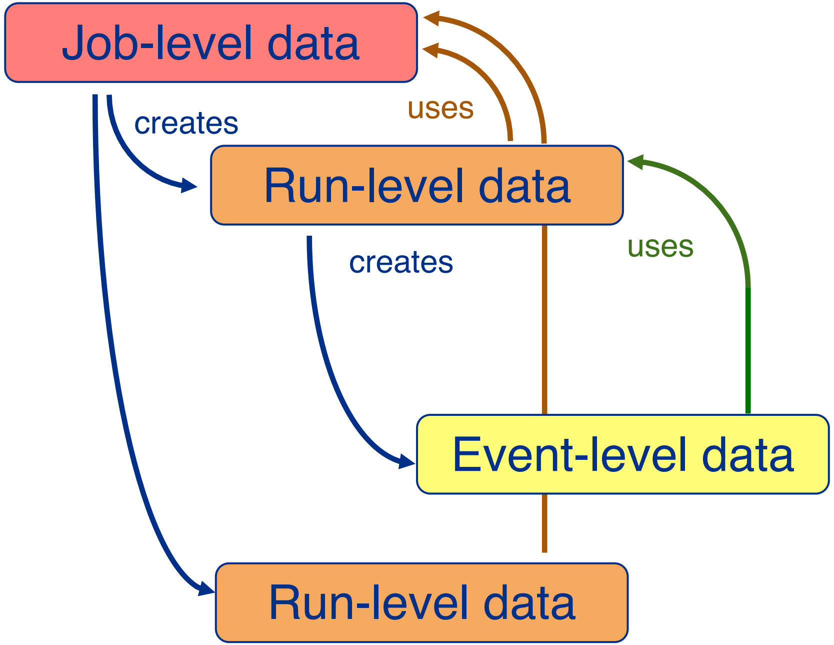
Thread 1	Thread 2
Process Event 1	
	Process Event 2

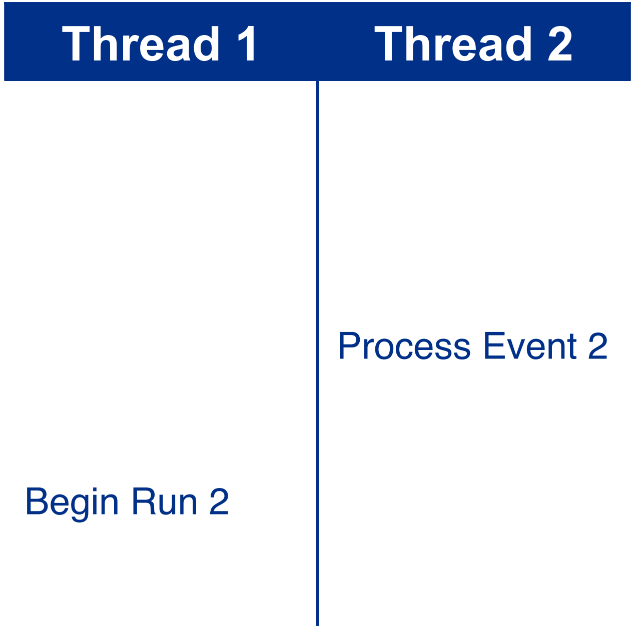
Persistent data structure approach



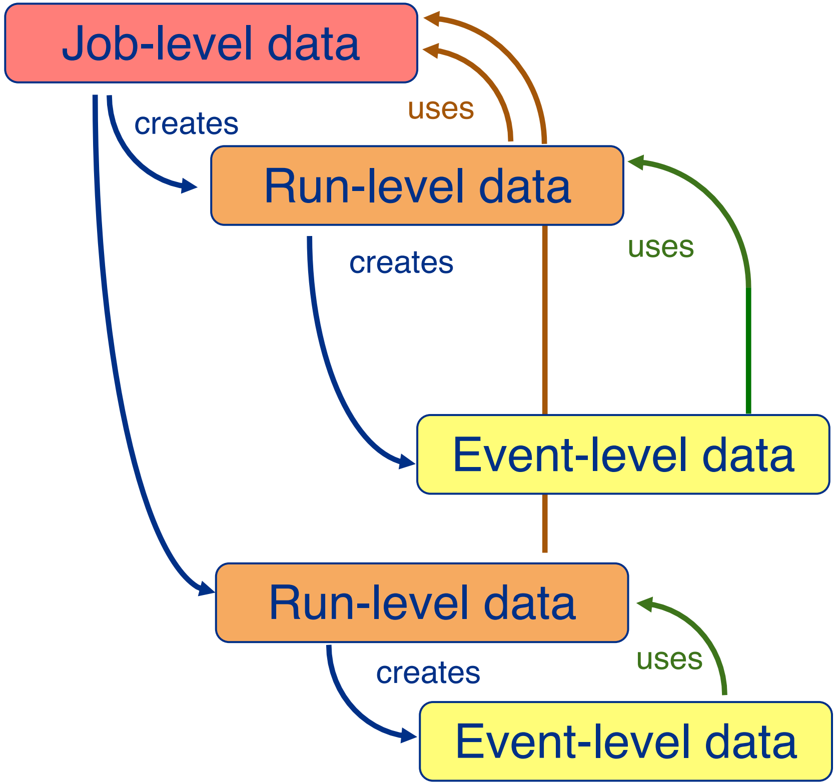


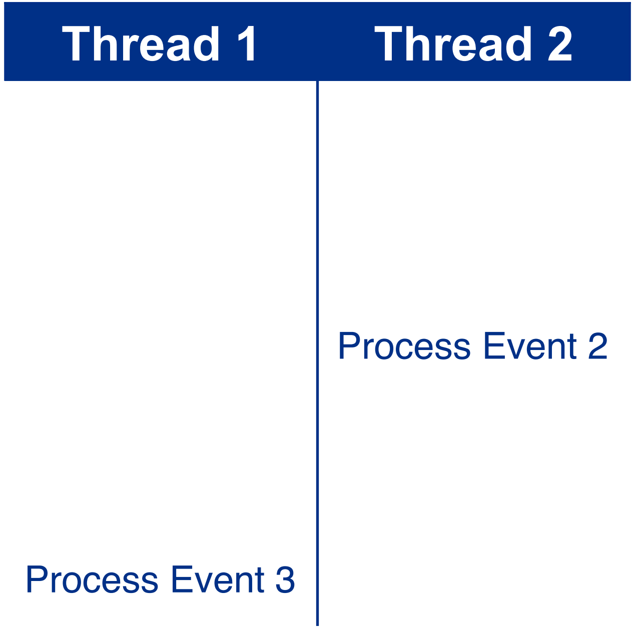
Persistent data structure approach



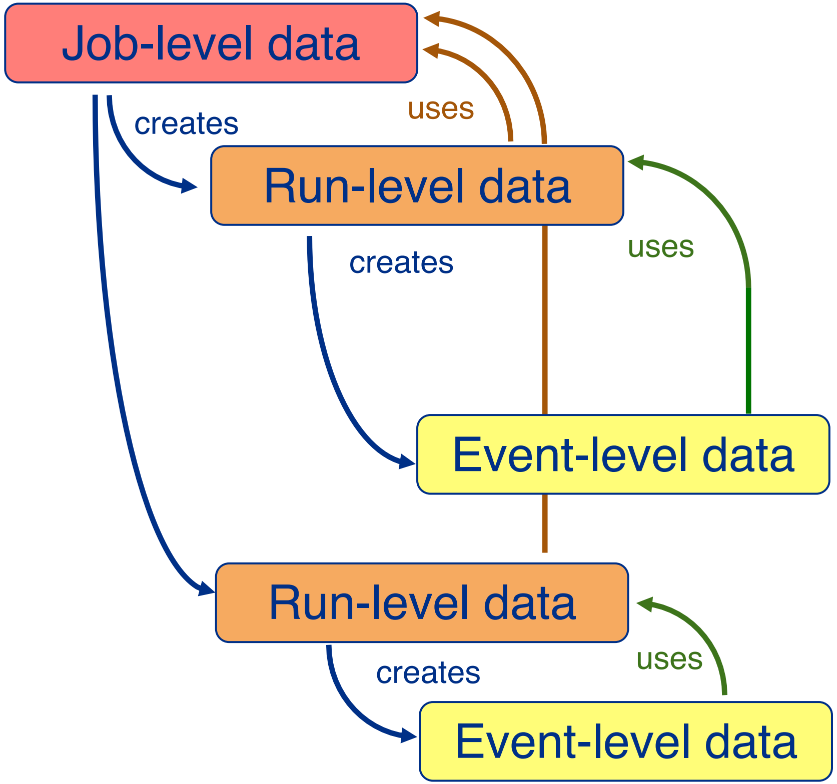


Persistent data structure approach



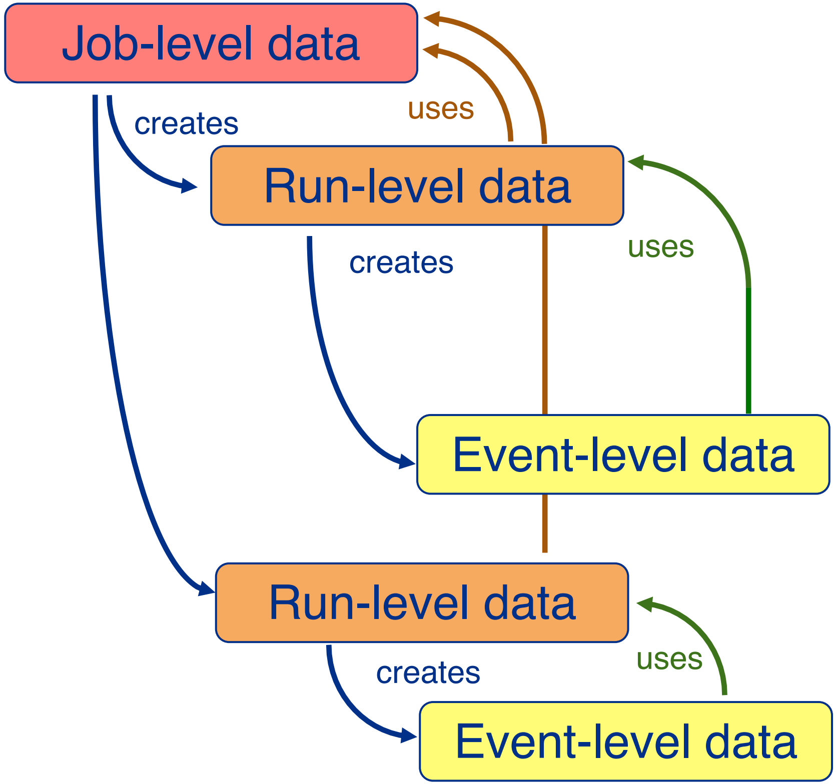


Persistent data structure approach



- **Why does this work?**
  - All objects are immutable.
  - Object construction/destruction happens on one thread.
  - Object of one processing level refers to the object directly above it (via pointer or reference).
  - Assuming data corresponding to each processing levels is small, extra overhead is minimal wrt. thread-unsafe option.

Persistent data structure approach



- **Why does this work?**
  - All objects are immutable.
  - Object construction/destruction happens on one thread.
  - Object of one processing level refers to the object directly above it (via pointer or reference).
  - Assuming data corresponding to each processing levels is small, extra overhead is minimal wrt. thread-unsafe option.
- **Downsides to this approach**
  - May require caching of data across threads. Not so much an issue for `DetectorClocks`.

Example: Thread-unsafe code

```
class ClockService {
public:
  ClockService(ParameterSet const& pset,
               ActivityRegistry& reg);

  string const& mode() const noexcept { return mode_; }
  RunNumber_t run() const noexcept { return run_; }
  Clock const* clock() const noexcept { return clock_.get(); }

private:
  void prepareRun(Run const& r);
  void prepareEvent(Event const& e, ScheduleID);

  string const mode_;
  bool goodRun_{false}; // Updated per run
  unique_ptr<Clock const> clock_{nullptr}; // Updated per event
};
```

Example: Thread-unsafe code

```
class ClockService {
public:
  ClockService(ParameterSet const& pset,
               ActivityRegistry& reg);

  string const& mode() const noexcept { retu
  RunNumber_t run() const noexcept { return run_; }
  Clock const* clock() const noexcept { return clock_.get(); }

private:
  void prepareRun(Run const& r);
  void prepareEvent(Event const& e, ScheduleID);

  string const mode_;
  bool goodRun_{false}; // Updated per run
  unique_ptr<Clock const> clock_{nullptr}; // Updated per event
};
```

```
ClockService::ClockService(ParameterSet const& pset,
                           ActivityRegistry& reg)
  : mode_{pset.get<string>("mode")}
{
  reg.sPreProcessRun.watch(this, &ClockService::prepareRun);
  reg.sPreProcessEvent.watch(this, &ClockService::prepareEvent);
}
```

Example: Thread-unsafe code

```
class ClockService {
public:
  ClockService(ParameterSet const& pset,
               ActivityRegistry& reg);

  string const& mode() const noexcept { retu
  RunNumber_t run() const noexcept { return run_; }
  Clock const* clock() const noexcept { retu

private:
  void prepareRun(Run const& r);
  void prepareEvent(Event const& e, Schedule

  string const mode_;
  bool goodRun_{false}; // Updated per run
  unique_ptr<Clock const> clock_{nullptr}; /
};
```

```
ClockService::ClockService(ParameterSet const& pset,
                           ActivityRegistry& reg)
  : mode_{pset.get<string>("mode")}
{
  reg.sPreProcessRun.watch(this, &ClockService::prepareRun);
  reg.sPreProcessEvent.watch(this, &ClockService::prepareEvent);
}
```

```
void
ClockService::prepareRun(Run const& r)
{
  goodRun_ = clock_is_valid_for(r);
}
```

```
void
ClockService::prepareEvent(Event const& e, ScheduleID)
{
  clock_ = get_clock(mode_, goodRun_, e);
}
```

Example: Thread-unsafe code

```
class ClockService {
public:
  ClockService(ParameterSet const& pset,
               ActivityRegistry& reg);

  string const& mode() const noexcept { retu
  RunNumber_t run() const noexcept { return run_; }
  Clock const* clock() const noexcept { retu

private:
  void prepareRun(Run const& r);
  void prepareEvent(Event const& e, Schedule

  string const mode_;
  bool goodRun_{false}; // Updated per run
  unique_ptr<Clock const> clock_{nullptr}; /
};
```

```
ClockService::ClockService(ParameterSet const& pset,
                           ActivityRegistry& reg)
  : mode_{pset.get<string>("mode")}
{
  reg.sPreProcessRun.watch(this, &ClockService::prepareRun);
  reg.sPreProcessEvent.watch(this, &ClockService::prepareEvent);
}
```

```
void
ClockService::prepareRun(Run const& r)
{
  goodRun_ = clock_is_valid_for(r);
}
```

```
void
ClockService::prepareEvent(Event const& e, ScheduleID)
{
  clock_ = get_clock(mode_, goodRun_, e);
}
```

Not everything is `const`. ☹

Example: Thread-safe code (using persistent data structures)

```cpp
class ClockService {
public:
  ClockService(ParameterSet const& pset)
    : mode_{pset.get<string>("mode")}
  {}

  string const& mode() const noexcept { return mode_; }

  class RunData;
  class EventData;

  RunData DataForRun(Run const& r) const;

private:
  string const mode_;
};
```

Example: Thread-safe code (using persistent data structures)

```
class ClockService {
public:
  ClockService(ParameterSet const& pset)
    : mode_{pset.get<string>("mode")}
  {}

  string const

  class RunDat
  class EventD

  RunData Data

private:
  string const
};
```

```
class ClockService::RunData {
public:
  RunData(string const& mode, Run const& r)
    : mode_{mode}
    , goodRun_{clock_is_valid_for(r)}
  {}
  string const& mode() const noexcept { return mode_; }
  bool goodRun() const noexcept { return goodRun_; }

  EventData DataForEvent(Event const& e) const;

private:
  string const& mode_;
  bool const goodRun_;
};
```

Example: Thread-safe code (using persistent data structures)

```
class ClockService {
public:
  ClockService(ParameterSet const& pset)
    : mode_{pset.get<string>("mode")}
  {}

  string const

  class RunDat
  class EventD

  RunData Data

private:
  string const
};
```

```
class ClockService::RunData {
public:
  RunData(string const& mode, Run const& r)
    : mode_{mode}
    , goodRun_{clock
  {}
  string const& mode
  bool goodRun() con

  EventData DataForE

private:
  string const& mode
  bool const goodRun
};
```

```
class ClockService::EventData {
public:
  EventData(RunData const& runData, Event const& e)
    : runData_{runData}
    , clock_{get_clock(runData.mode(), runData.goodRun(), e)}
  {}

  RunData const& runData() const noexcept { return runData_; }
  Clock const* clock() const noexcept { return clock_.get(); }

private:
  RunData const& runData_; // By reference to avoid large memory
  unique_ptr<Clock const> const clock_;
};
```

Example: Thread-safe code (using persistent data structures)

```
class ClockService {
public:
  ClockService(ParameterSet const& pset)
    : mode_{pset.get<string>("mode")}
  {}

  string const

  class RunDat
  class EventD

  RunData Data

private:
  string const
};
```

```
class ClockService::RunData {
public:
  RunData(string const& mode, Run const& r)
    : mode_{mode}
    , goodRun_{clock
  {}
  string const& mode
  bool goodRun() con

  EventData DataForE

private:
  string const& mode
  bool const goodRun
};
```

```
class ClockService::EventData {
public:
  EventData(RunData const& runData, Event const& e)
    : runData_{runData}
    , clock_{get_clock(runData.mode(), runData.goodRun(), e)}
  {}

  RunData const& runData() const noexcept { return runData_; }
  Clock const* clock() const noexcept { return clock_.get(); }

private:
  RunData const& runData_; // By reference to avoid large memory
  unique_ptr<Clock const> const clock_;
};
```

Everything is `const`*.* ☺

But what does `DetectorClocks` look like?

- As only events within a subrun can be processed concurrently at the moment, only event-level data must be protected.
- The majority of the `DetectorClocks` interface still exists, but there is an extra layer in between called `DetectorClocksData`.

But what does DetectorClocks look like?

- As only events within a subrun can be processed concurrently at the moment, only event-level data must be protected.
- The majority of the DetectorClocks interface still exists, but there is an extra layer in between called DetectorClocksData.

Old interface

```
using detinfo::DetectorClocksService;

MyProducer::MyProducer(ParameterSet const& pset)
{
  ServiceHandle<DetectorClocksService const> clocks;
  double beam_time = clocks->BeamGateTime();
}


void MyProducer::produce(art::Event& e)
{
  ServiceHandle<DetectorClocksService const> clocks;
  double beam_time = clocks->BeamGateTime();
}
```

But what does DetectorClocks look like?

- As only events within a subrun can be processed concurrently at the moment, only event-level data must be protected.
- The majority of the `DetectorClocks` interface still exists, but there is an extra layer in between called `DetectorClocksData`.

Old interface

```
using detinfo::DetectorClocksService;

MyProducer::MyProducer(ParameterSet const& pset)
{
  ServiceHandle<DetectorClocksService const> clocks;
  double beam_time = clocks->BeamGateTime();
}


void MyProducer::produce(art::Event& e)
{
  ServiceHandle<DetectorClocksService const> clocks;
  double beam_time = clocks->BeamGateTime();
}
```

New interface

```
using detinfo::DetectorClocksService;

MyProducer::MyProducer(ParameterSet const& pset)
{
  ServiceHandle<DetectorClocksService const> clocks;
  auto const clockData = clocks->GlobalData();
  double beam_time = clockData.BeamGateTime();
}

void MyProducer::produce(art::Event& e)
{
  ServiceHandle<DetectorClocksService const> clocks;
  auto const clockData = clocks->DataForEvent(e);
  double beam_time = clockData.BeamGateTime();
}
```

Consequences of this change

- Code using the `DetectorClocks` service must know if event-level data or global data is needed.
  - There are cases in the code where global-level data is cached by a module, and then used along with event-level detector-clocks values.

- Framework-agnostic code that creates a `DetectorClocks` service handle must be adjusted to receive the correct information.

- Sounds like a big change (it is!), but there are upsides to it:
  - I've implemented the majority of the changes on feature branches.
  - There are no new run-time dependencies; the dependence on `DetectorClocks` is just more explicit, and thus clearer.
  - In some cases, dependence on `DetectorProperties` was removed.
  - This gets us closer to multi-threaded execution of LArSoft facilities.

Proposal

- **Proposal:** The "persistent data structures" approach should be adopted for the `DetectorClocks` and `DetectorProperties` providers and services.

- Current status
  - All LArSoft repositories have `feature/team_for_mt` branches using the new interface.
  - I am working on adjusting experiment repositories' use of `DetectorClocksService`.
  - I suggest merging the `feature/team_for_mt` branches ***after the move*** to GitHub. This will allow the design to solidify, possibly enabling me to adjust the `DetectorProperties` interface before then.
  - ***FYI:*** *Due to large number of changes, I have applied clang-format to those files that required adjustment.*

- I will present the list of breaking changes once the feature branches are ready to go to GitHub.