Reviewing Athena I/O Components from the Thread Safety Perspective

Marcin Nowak, BNL

Thread Safety Concepts

- Code can be made thread-safe to a different degree

1. „True” thread safety requires data structure integrity protection on different levels, correct execution logic and deadlock avoidance
   - Good parallelism, low memory overhead
   - (Very?) difficult to implement
2. Global lock and access serialization
   - No parallelism but also no memory overhead
   - Easy to implement
3. Concurrent independent workers
   - Good parallelism but biggest memory overhead
   - Ease of adaptation of an existing code depends on the design
     - Requires code review to find places where workers interact with each other
     - Threat: singletons (global, static, framework-wide data structures)

ROOT Thread Safety

- ROOT thread safety is continuously improving
  - we should be using at least ROOT 6.4
  - ROOT 6.6 adds improvements in usability
- Important points:
  - ROOT „static” API (TClass, gSystem, gInterpreter) is thread-safe
    - Important for types and dictionaries
  - I/O operations can be performed simultaneously on **different** files
    - The same file can not be opened twice (without closing it first)
    - The same file can be accessed by different threads in a serialized manner
  - We have not tested these features yet!

- APR opens ROOT files using FileManager
  - Read-only files are shared
  - Still need to check thread safety

Athena I/O Components

- Core APR components:
  - PersistencySvc
  - StorageSvc
  - RootStorageSvc
- Helper APR components
  - FileCatalog
  - Collections
- AthenaPOOL
  - PoolSvc
  - AthenaPoolCnvSvc
  - T/P Converters
- Other
  - FileManager

APR: RootStorageSvc

- Implementation of the StorageSvc using ROOT persistency
- Granularity – ROOT file
  - Corresponds to the notion of an APR database
  - May have multiple TTrees
    - Most of ATLAS Event Data in one common tree
  - Multiple object containers
  - Single transaction per file (database)
- Original POOL code seems well adapted to the concurrent workers model
  - No global or static data
  - Possibly needing a cleanup from ROOT5 features: ClassLoader
- Recent improvements for xAOD introduced global AuxTypeRegistry
  - Believed to be made thread-safe by Scott
- FileManager – to be checked

APR: StorageSvc

- This package deals with datastore organization (databases, containers), transactions and type information
- Not really a service – multiple instances can be (and are) created
  - good fit for the concurrent worker model
- Several instances of static data:
  - Debugging counters
    - Used only for special purpose debugging, otherwise thread-safe
    - May need to be protected
  - Central object schema repository
    - Needs to be made thread-safe
  - Type/GUID maps
    - Needs to be made thread-safe
- Static data needs to be protected in any scenario involving parallel I/O

APR: PersistencySvc

- Our top-level APR package dealing with file catalog, configuration, database registry and managing the global transaction
- Also not a service – Athena already uses multiple instances of it
  - In fact it's just a wrapper around UserSession class
- No global or static data
  - But the FileCatalog accesses a file on disk, which makes it a shared resource

- Creating an instance of PersistencySvc automatically creates a new StorageSvc and the required RootStorageSvc objects. These instances can be used simultaneously, as long as they all open different files.
  - Assuming we add protection for StorageSvc static data and check FileCatalog and FileManager

T/P Converters

- T/P converters are standalone conversion objects that are used mainly by the AthenaPOOL layer (above APR)
  - A T/P converter is usually a data member of an AthenaPOOL (Gaudi) converter that is managed by the Conversion Service
  - The preferred way to use T/P converters concurrently would be to create multiple copies of them (the concurrent worker model)
    - but that is not easy with the current Conversion Service

- Early test with AthenaHive with simultaneous reading and writing showed success when 2 separate instances of the AthCnvSvc were used – one for reading and another one for writing
  - Each instance of AthCnvSvc creates separate copies of converters