Making ROOT I/O Thread-Safe

Christopher Jones *FNAL*

Goal

Make it safe to

Read multiple TFiles on different threads
Write multiple TFiles on different threads
Have calls to other ROOT functions on other threads not interfere with I/O

Not a goal

Reading/writing one TFile on multiple threads

Tools

Known problems

List from Philippe Canal
List of shared resources

Valgrind's helgrind tool

Finds data races and mutex ordering problems

Static analysis

Written using LLVM
Builds a graph of which functions call which other functions
Finds all globals
Creates a list of all functions which connect to a global/shared resources

Procedure

Fix known problems

Use static analyzer to find connects between I/O routines and known shared resources

Protect those resources

Build test system

Start N threads

On each thread open a TFile for reading and one for writing then copy all objects from one to the other

Run helgrind on test system

Unexpected Connections

Found several unexpected connections between components

Filling a TH1 can interfere with opening a TFile

- TH1::Fill can cause a rebinning of the histogram
- Rebinning calls TObject::Clone
- TObject::Clone uses serialization code
- Serialization code does lazy work
- Lazy work can also happen when opening a TFile

Added protections for these connections

Types of Fixes

Algorithm changes

C++11 thread_local

C++11 std::atomic<>

Adding more mutex locks

Algorithm Changes

TObject determining if instance on stack or heap

Previously kept a begin and end value for addresses given from heap

- If new object this between values assumed to be on heap

Heap in threaded program not contiguous

- Stacks for new threads can exist between heap sections

Changed to TObject::new causing memory to be filled with 'magic' value

- If member data set to 'magic' value then on heap

No need for global info (begin and end values) so no thread problem

Avoid rebuilding StreamerInfos

Use thread-safe flag in object to denote work has already been done
Also speeds up single-threaded case

Avoid resetting values

Check if a value is not already what you are going to change it to

Using C++11

Only C++11 has a memory model for threading

Only C++ version which gives portable threading

Decided with Philippe that ROOT would only be thread-safe when compiled using C++11

std::thread_local

Used for globals in ROOT which hold temporary state for a callstack

E.g. TClass::fgCallingNew only used during callstacks involving TClass::New

std::atomic<>

Used for global variables used to assign unique IDs

Used for member data which are caches

mutex

Originally incomplete coverage of mutex in ROOT code

often mutex used to lock change in structure but no lock when reading

Used static analysis to determine best routine to make lock

Helgrind used to find lock ordering problem

If have two mutex (A,B) and two different threads take the locks indifferent order (A then B vs B then A) can lead to deadlock

Results

Test program now runs with no reported helgrind errors

Performance

As number of threads increased the event throughput was constant

Found that I/O was completely serial

gdb 'polling' showed N-1 threads were stopped in mutex wait

Code

Available on github

https://github.com/Dr15Jones/root

Being used as basis for Philippe Canal's threading work