Dealing with I/O in Multithreaded Environments

Marc Paterno

Fermilab Computing Division
CET group

Multicore Programming Mini-workshop
October 10, 2008

Multithreading may complement fork/copy-on-write

- The announced topic of this workshop was multiprocess programming and the fork/copy-on-write technique.
- My primary goal is to remind us why multithread programming, at several levels of granularity, is worthy of continued investigation.
- My secondary goal is to point out some places where common tools defeat our efforts at multithread programming, in the hope that we can find some common solutions.

Making efficient use of resources: fork/copy-on-write

- The simplest use of a multicore computer is to use it as if it were several independent computers.
- Because the memory footprint of our "typical" event-processing processes is large, in part because of large ancillary data.[1], we may exhaust memory before we exhaust CPU power.
- For coarse-grained parallelism[2] the fork/copy-on-write technique can help, when we share much ancillary data.
- But when data are not shared (or when they become unshared, and copying happens), we lose the benefit. In the limit of no sharing, no benefit is gained. There are some important uses that approach this limit:
  - when processing skim data, in which each event may come from a different run;
  - when there are few events—or only one—to be processed, which is common during software development.

[1] non-event data; *e.g.* detector geometry and calibration constants.

[2] concurrent processing of independent events.

Making efficient use of resources: multithread

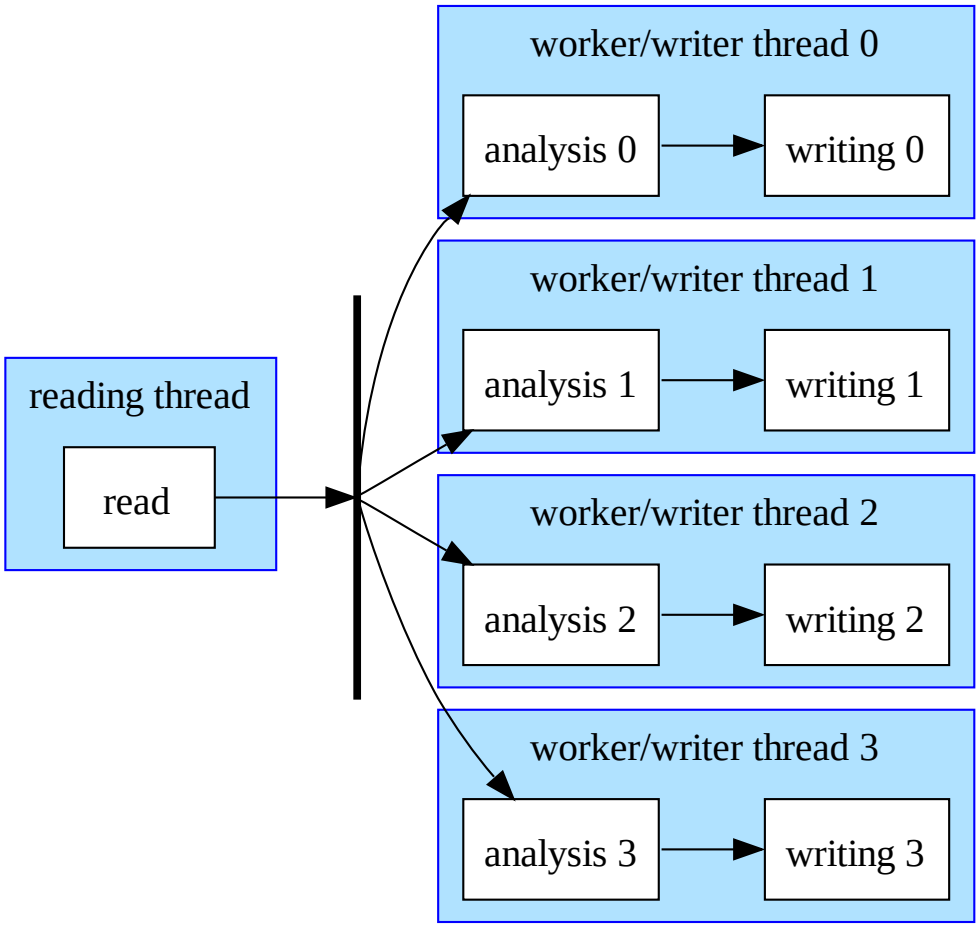
- Multithread programming has different advantages than multiprocess programming. It allows a variety of granularities.
  1. Event-level (coarse-grained) multithreading is similar to fork/copy-on-write; it is beneficial in the same circumstances, and loses its benefits in the same circumstances.
  2. We can allow the *event processing framework* to schedule concurrent work on the same event (medium-grained multithreading).
  3. We can allow *individual framework modules* to schedule concurrent work on the same event (fine-grained multithreading).
  4. Items 2 and 3 do not suffer from memory exhaustion due to ancillary data.
- We can use multiple granularities simultaneously, perhaps optimizing each to adapt to the specific machine and job.
- Perhaps we can benefit from clever use of local memory on each core. We can leverage Fermilab's local expertise (primarily in the LQCD community) in NUMA.

Making efficient use of resources: concurrent i/o

- Modern machines frequently support multiple i/o channels, and have networked disks.
- With such machines, we can often benefit from concurrent i/o. Especially in multithread applications, we can:
  - read events, and perhaps uncompress or preprocess them, independently of the real processing work;
  - write events independent of reading them; and
  - write events to different output streams simultaneously.
- What can we learn from the experience of others in the scientific community (HDF5, parallel netCDF, MPI-IO)?
- Concurrent i/o may help to optimize our use of *both* CPU resources *and* i/o resources.

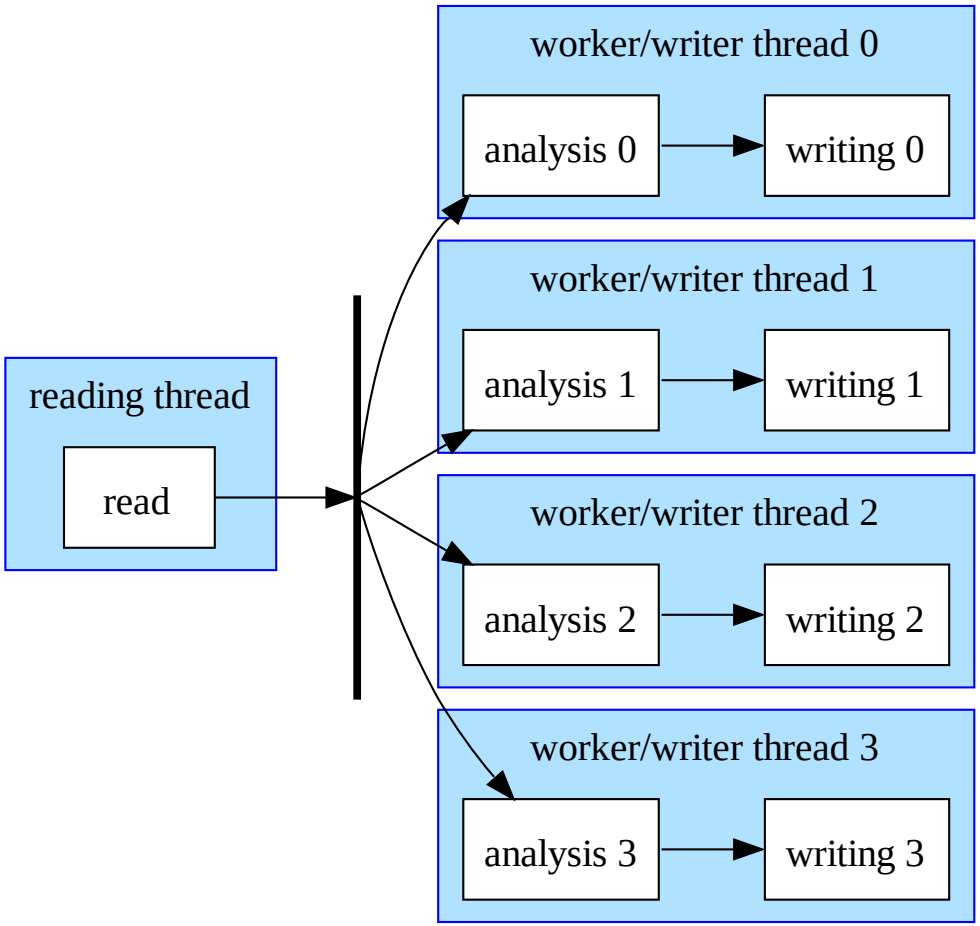
A real user's (failed) multithreaded ntuple writing program

Each event is given to every worker.



A real user's (failed) multithreaded ntuple writing program

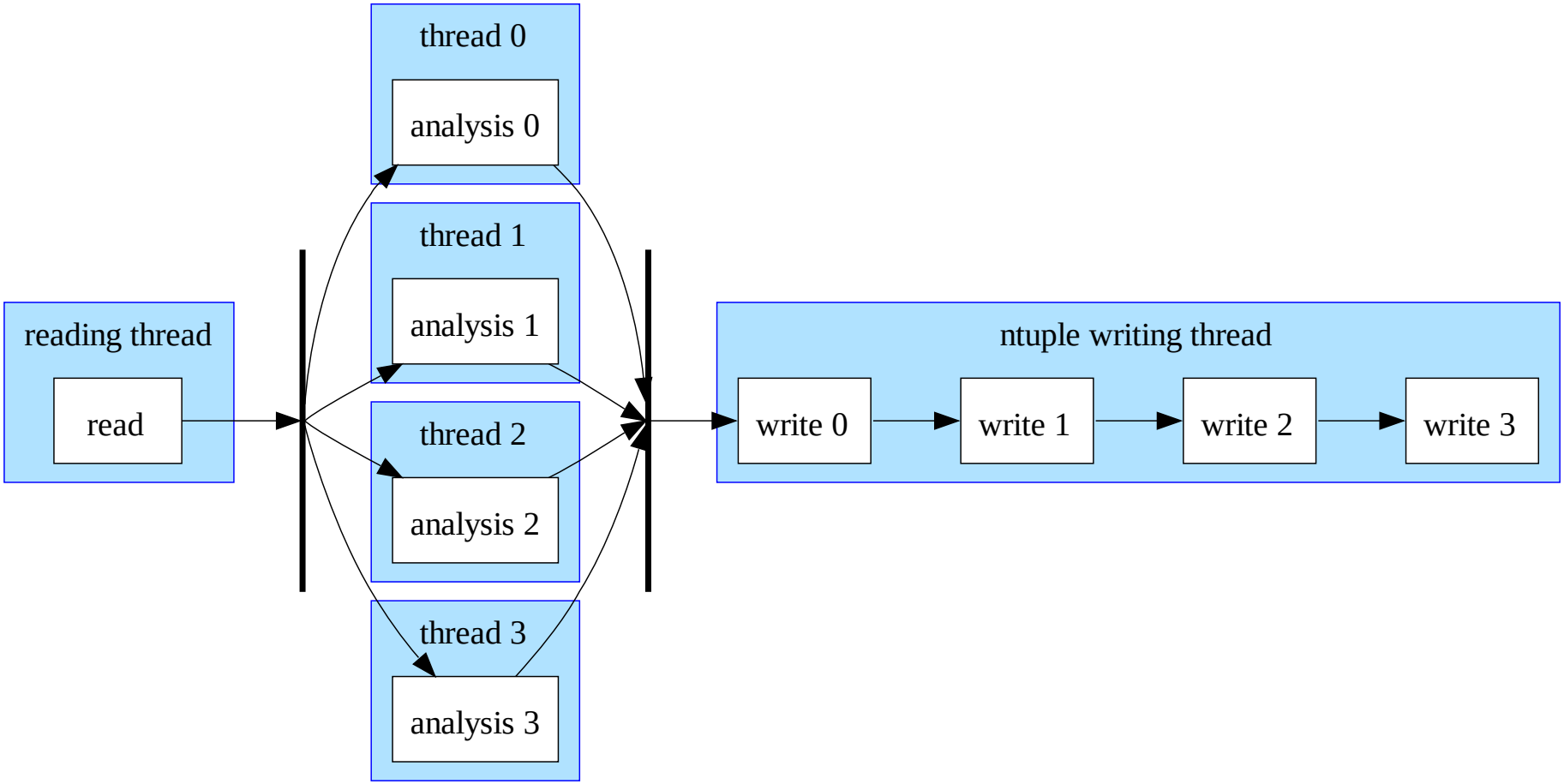
Each event is given to every worker.



- This user wanted to run several (actually 8) analyses simultaneously, each one filling independent histograms and writing an independent ntuple.
- The program would run for a fraction of a second and crash; our group was asked to help diagnose the problem.
- Failures happened both in creating and filling histograms and in writing the ntuples.

The modification needed to make the program run

Each event is given to every worker. I/O bottleneck from serialization.



User wrote his own simple histogram code that was thread-safe.

CMS would benefit from concurrent I/O

- The CMS event-processing framework, and its driver `cmsRun`, were designed for easy extension to concurrent event processing.
- We intended to support coarse-grained or medium-grained parallelism.
  - Use the *producer-consumer* model for concurrency.
  - One **EventProcessor** would manage an **InputSource**, running in its own thread, and several worker threads each running all (or some of) the user-configured paths, and several output threads, each running a single **OutputModule**.
  - The single **InputSource** would be the "producer" of events.
  - Events would be passed to a worker thread as soon as one became available.
  - Events would be written to output as soon as they had finished the worker threads.
- This solution requires that the I/O subsystem support concurrent reading and writing, and multiple simultaneous writers.
- This is not the only example of where CMS would benefit from concurrent I/O.

Concurrent serialization and deserialization

- We read parts of each event from many **TBranch**es, spread across several **TTree**s.
- We write events similarly.
- To avoid I/O bottlenecks we would like to read concurrently while writing.
  - Events being written are independent of that being read.
  - Global locking that serializes all reading and writing is not acceptable; the I/O bottleneck would render many programs unscalable.
- We write (possibly different) parts of the same event to different output streams.
  - We want to write the streams concurrently.
  - We must be able to write the same event in different output modules.

Concurrent creation, filling and drawing of histograms

- Module instances (**EDProducer**s, *etc.*) in different threads need to independently create and fill their own histograms.
- Histogramming code need not (and *should* not) provide object-level locking; such designs often are inefficient. Client code should assure safety of shared objects.
- Library code should *not* share objects "behind the scenes" in a fashion that prevents concurrent use. Coarse-level serialization that defeats the purpose of user-level concurrency is not an acceptable solution.
- GUI programs must also support display-driver threads that paint objects on the screen simultaneously with the other work.

Simultaneous reading, writing, and histogramming

- We need to create and fill histograms in the same program that is performing concurrent I/O.
- A full solution to our concurrency programs should allow concurrent reading of one event, writing of multiple events each to multiple output modules, concurrently with creation and filling of histograms which are *not* shared between threads.
- This solution can not be obtained by coarse-grained locking that defeats the concurrency goals of the design.

Summary

Multithreaded programming may complement a fork/copy-on-write strategy for efficient use of multicore processors.

Multithreaded programming may provide benefits in conditions where fork/copy-on-write does not help.

- Processing samples with little shareable data, *e.g.* skim data samples.
- Processing single events, *e.g.* debugging.

Concurrent i/o is valuable, especially in multithreaded applications.

We should pursue multithread support in our common libraries. My group at Fermilab hopes to pursue this within GEANT4.