ARC Centre of Excellence for Autonomous Systems

C++ Threading

Tim Bailey

Introduction

- Very basic introduction to threads
- I do not discuss a thread API
  - Easy to learn
    - Use, eg., Boost.threads
    - Standard thread API will be available soon
  - The thread API is too low-level for thinking about multi-threaded programming
    - Your program will be complicated and error-prone

What are Threads?

- Multiple simultaneous paths of execution
  - Single processor; pre-emptive time slicing
  - Multiple processors; true parallel processing

- Shared address space
  - Threads can access the same pointers and objects

Why Threads?

- Some programs have a simpler, more natural design with threads
- Some programs are more responsive

- “Why?” is not what this talk is about

Essence of Multi-Threaded Design

- Threading is about *communication*
  - Passing/sharing data between threads
  - Race conditions
  - Deadlocks

- Good design makes multi-threaded code look like single-threaded code
  - Write code where you don't have to think about threading issues

Shared Objects and Race Conditions

- Writing to a shared state
  - Non-atomic actions can cause tricky problems
- Complications, eg., cache
  - See double checked locking debacle

  http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html
- Simple solution:
  - Put mutexes around ***ALL*** accesses to shared objects

Mutexes and Locks

- A mutex is a ***shared*** object associated with shared data
  - [see mutex.cpp, wrapping pthreads]
- A lock is a ***local*** object associated with a local critical section
  - Lock and unlock a mutex around a code block that accesses the shared data
- Unlocking can be tricky
  - Programmer forgetfulness
  - Exceptions and resource release
  - Solution RAII [see scopedlock.cpp]

Typical (Bad) Design

- Manual locking [see scopedlock.cpp]
- Problems
  - Must remember to lock every time
  - Must associate correct mutex with the data
  - Threading is mingled throughout the code
- All code looks “multi-threaded” and is therefore coupled and complicated
  - Tricky thread-oriented debugging is not localised

Enforcing Shared Object Locking

- Monitor Object [see monitorobject.cpp]
  - Non-intrusive ***internal*** locking
  - Use to lock around ***individual*** member accesses
  - Issue: multiple member accesses are not atomic

- Accessor Object [see sharedobject.cpp]
  - Enforced ***external*** locking
  - Use to lock around ***multiple*** member accesses

Condition Variables

- Condition variables
  - Wait condition, allows a thread to block
  - Signal from another thread unblocks waiting thread; the two threads are synchronised
  - [see condition.cpp, wrapping pthreads]

- Condition variables are an essential component for building communication channels

Communication Channels

- Unidirectional channels
  - Thread safety encapsulated within channel
  - Permits localised thread debugging
  - Interface looks like single-threaded code

- Channel has sender and receiver perspectives
  - A thread sees only one endpoint
    - Send endpoint, OR
    - Receive endpoint
  - Use proxy interfaces to enforce correct access
  - [see channel.cpp]

Caveat: Memory Allocation

- Use of ***new*** invokes a global mutex
  - Synchronises threads, degrades performance
  - Worst if execution is parallel, not just time-sliced
- A problem for **all** heap-based allocation
- Advice:
  - Consider using pre-allocated memory pools
  - Use fixed-size channel buffers
  - Avoid communicating objects whose constructors allocate dynamic memory, since
    - Blocking channel makes one copy
    - Buffering channel makes two copies

What Remains?

- Lots...

- Read-write locks, try locks, spin locks, timeouts
- Re-entrant functions; avoid global and static variables
- Deadlock prevention/detection
- Thread priorities and scheduling
  - Priority inversion
- Patterns
  - Producer-Consumer
  - Thread pools
  - Active objects with Futures
  - Publish-Subscriber
- Thread lifetimes
  - Thread termination; graceful
  - Detached, non-detached, joining
  - Threads and exceptions
- Channel varieties
  - Blocking, buffering, one-to-many, many-to-one
- Name services
  - IDs of active threads, classes and objects
  - Dynamically connecting/disconnecting channels at runtime
  - Connection requests/notification

Conclusion

- Isolate thread-aware code
  - Localise thread debugging to a few classes
- Majority of code communicates through high-level interfaces
  - Interfaces encapsulate thread safety
  - Most code is written as single threaded (ie., thread-oblivious)
- Simplicity is key