Debugging Multi-threaded Programs

Designing for debugging

- Use proper software development practices
  - Don't code first without designing first
  - Design with testing in mind
- More crucial for multi-threaded applications
  - multi-threaded applications are inherently more complicated
    - Large number of special cases
    - Wide range of possible paths
    - Access patterns may vary widely between single and multi-core
  - Multi-threaded bugs may not appear when running under debugger
    - Bugs sensitive to timing of events

Points to keep in mind

- Design the application so it can run sequentially
  - Validate in this mode first
- Use established parallel programming patterns
  - Solve many common problems
- Include built-in debug support in the application
  - Useful to be able to examine state of system at arbitrary time
  - Add functions to display state of thread or all threads
  - Use trace buffers to record sequence of accesses to shared resource

Do Code reviews

- Reviews challenging for parallel programs
- One technique
  - Individual reviewers examine from perspective of one of threads
    - Step through events as thread would
    - Have reviewers take and release shared resources
- Why validate?
  1. Take a few weeks up front to validate and verify the design
  2. Have to redesign from scratch when doesn't scale
- Why review?
  - Do you want to be debugging unpredictable bugs a week before the demonstration

Trace Buffers for Debugging Synchronization Bugs

- Need 2 pieces of information
  1. Which threads are accessing resource at time of failure
  2. When access took place
- A log or trace of the access pattern of different threads helps narrow down code to be reviewed
  - Trace buffer one way to do this
  - Mechanism for logging events
  - Uses atomic counter to keep track of empty slot in event log array
  - Information stored up to developer

Windows Trace Buffer example

- Windows app - progs\Ahkter\Ch8\DeadlockDebugApp.cpp
- Trace buffer stores 1024 events in circular buffer
  - Atomic index wraps
  - Don't need to dynamically resize buffer
- traceBufferElement is the event descriptor
- Three operations implemented
  - InitializeTraceBuffer
    - Initialized atomic counter to -1
  - AddEntryToTraceBuffer
  - PrintTraceBuffer
    - Useful if debugger allows execution of code at breakpoint
    - GDB and Visual Studio allow this
    - Can see recent events

Issues with trace buffers

- Logs events as passed into buffer
  - Does not guarantee that will log events exactly as they occur in times
  - Can have data races between thread writing shared variables and trace buffer
  - Example

Thread1	Thread2
m_global=dowork();	
	Thread_local_data=m_global;
	AddEntryToTraceBuffer(msg);
AddEntryToTraceBuffer(msg);	

  - Buffer does not reflect sequence of events
  - Can solve by placing locks around event and related log to trace buffer

Drawbacks to protecting logging

- May mask synchronization problems in original code
- Have protected the critical section that was unprotected
- When tracking down race condition, should avoid synchronization to access buffer
  - Preferred method is to log before and after event
    - If before and after messages occur in order can assume the event ordered
    - If before and after messages are interleaved then the order of events is indeterminate
- In general if you protect/synchronize access and application works, probably a problem with synchronization in the original code

Debugging using gdb

- Gdb capabilities for multi-threaded debugging
- Not all versions support all features
  - Automatic notification when threads created
    - displays threadid called ***systag*** - OS identification of thread
  - List all threads
    - **info threads** will print list of threads giving GDB thread number, systag, current stack frame, indicates active thread with *
  - thread-specific breakpoints
    - **break *linespec* thread *threadnum***
    - Stops all threads
    - Can have conditional break-points e.g.
      break buffer.c:33 thread 7 if level > watermark

Debugging using gdb

- thread-specific breakpoints - other issues
  - System calls may return early when a breakpoint is triggered
    - GDB uses signals to manage breakpoints, cause system calls to return
  - Advisable to check return values of sys calls and handle this case
    - Ex: sleep(30) syscall may return early – return value will be number of seconds left to sleep
  - GDB does not singlestep all threads in lockstep
    - may execute lots of code in other threads
    - If breakpoints in other threads may jump to these
    - Some versions of GDB have scheduler locker – only current thread can run

Debugging using gdb

- Ability to switch between threads
  - Use **thread threadnum** command
- Ability to apply commands to group of threads
  - Use command **apply**
  - argument can be threadnum or keyword **all**