
# Unpicking the weave:

## A practitioners view of challenges faced in debugging multithreaded/ many-core applications

**Neil Hickey**
**Samsung Research**
**Institute: UK**

# Samsung Research UK

- Research Institute focused on developing innovative software solutions for mobile, web and smart TV.
- SRUK are heavily involved in Khronos standards group, leading the UK Khronos chapter and pushing development of open standards for graphics/ compute APIs
- SRUK specializes in optimisation and debugging of Android software utilising the CPU and GPU (OpenGL ES, OpenCL)

# Multithreaded debugging

- **Why multi-threaded.**
  - **SRUK optimise applications making use of the GPU. This is typically a single application utilising the GPU and multiple threads of the CPU simultaneously.**
- **Multicore / Many core**
  - **Debugging multi-threaded CPU code is difficult enough.**
  - **Debugging GPU code is an order of magnitude more difficult**

# Multi-threaded development challenges

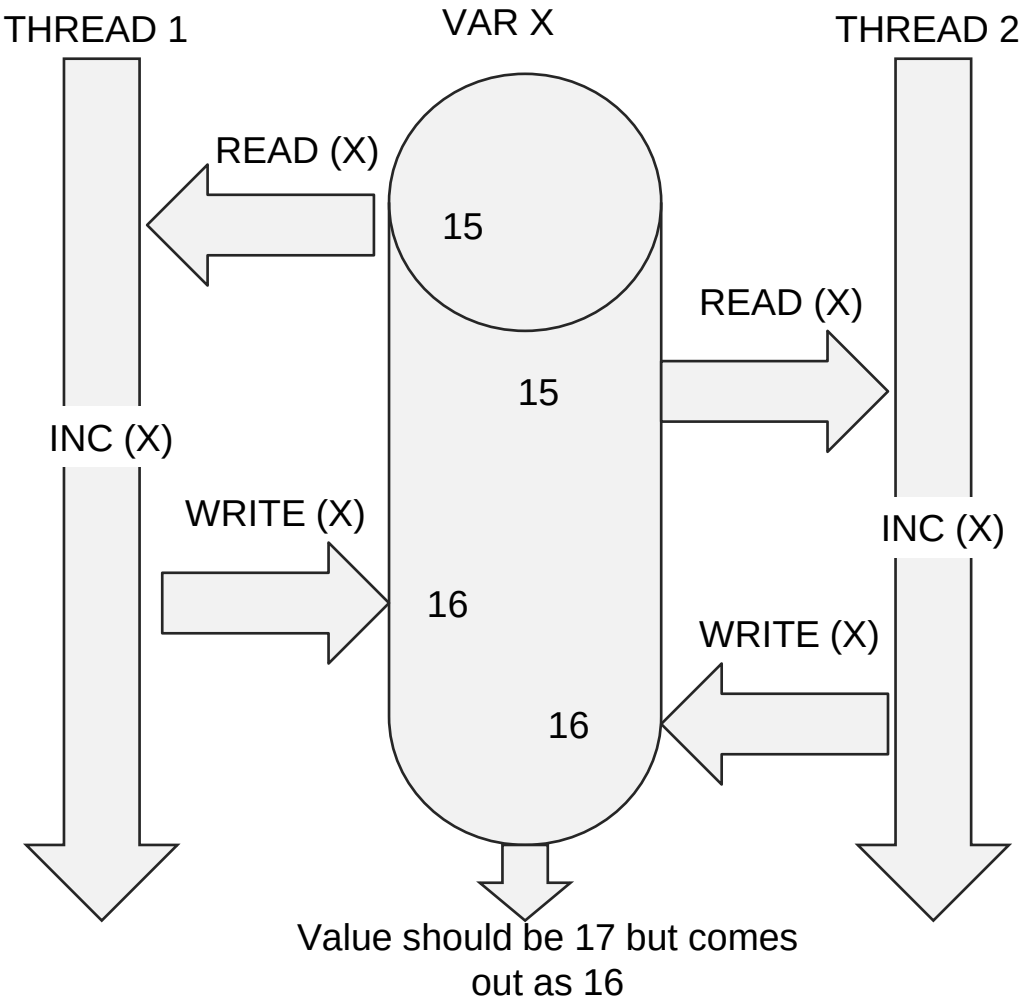
- **It is difficult.**

# Problems of parallel programming

- Data races
- Deadlock / Live lock
- Serialisation
- Simultaneous memory updates

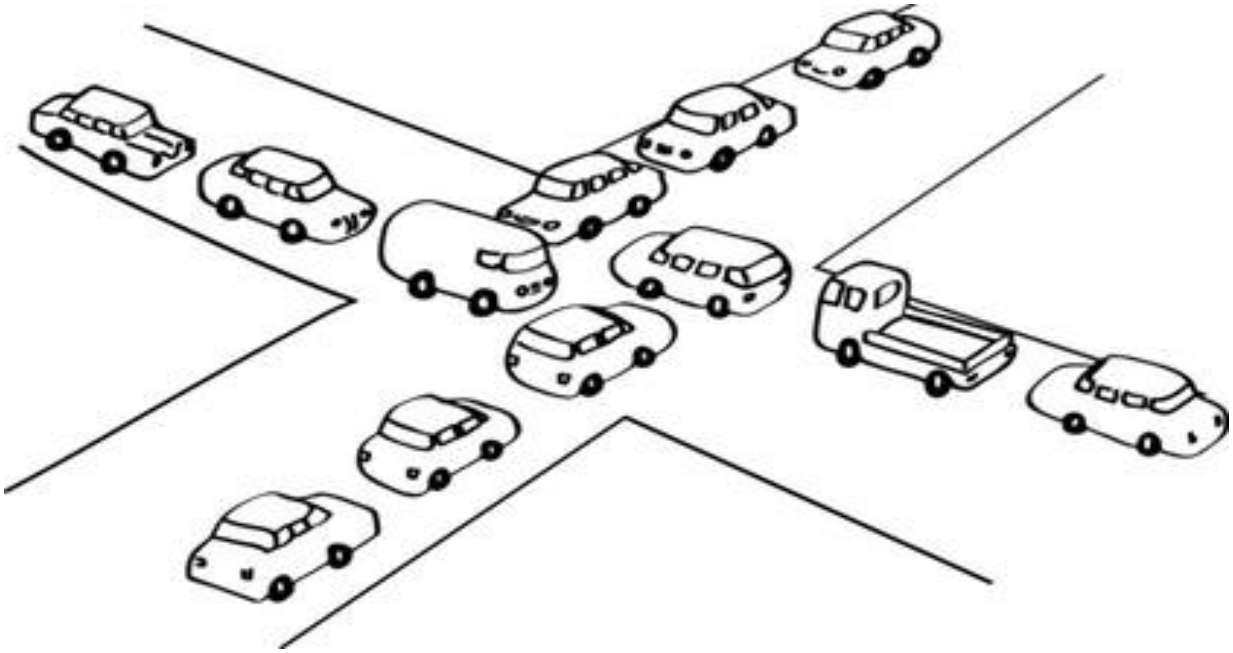
# Data races

- Multiple threads attempting to access the same piece of data.
- Non parallel behaviour while another thread is updating.



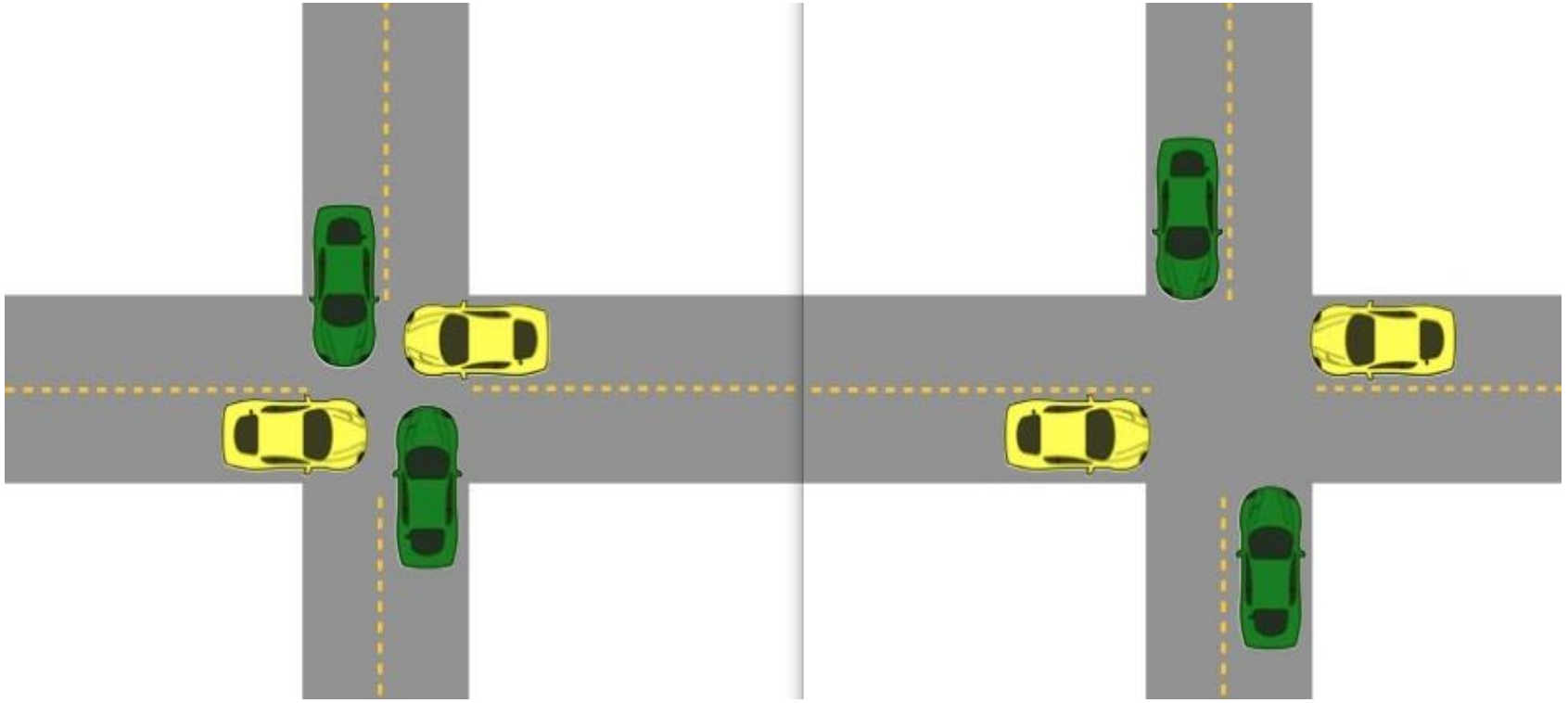
# Deadlocks

- Multiple threads attempting to secure a lock.
- Two threads each capture a lock on a resource and freeze up attempting to capture the lock on the other threads resource



# Live locks

- Two locks needed, one thread gets one, another thread gets another. Each thread tries to get the other and when it fails, frees its own lock.



# Serialisation

- **Multiple threads accessing shared resource, serialising on that shared resource therefore running no better than synchronous.**

# Simultaneous memory updates

- Multiple threads accessing data, e.g. Free pointers and causing data corruption.

# Multi-threaded debugging tips and tricks

- Trace buffers
- Debugging optimised code
- Use asserts and capture core files
- Perform object hand-over rather than shared objects
- Built-in debug support
- RAII for locking/unlocking mutexes
- Conditional breakpoints

# Trace buffers

- Instrument interesting sections of code with trace buffer writes to allow a textual representation of the access order.

  Example

```
int Thread1()
{
   ...
   TraceEvent(beforeMsg);
   doSomething();
   TraceEvent(afterMsg);
   ...
}

int Thread2()
{
   ...
   TraceEvent(beforeMsg);
   doSomethingElse();
   TraceEvent(afterMsg);
   ...
}
```

# Trace buffers example output

```
Trace 1 : StartWork Thread 1  time 79ms
Trace 2 : EndWork Thread 1  time 83ms
Trace 3 : StartWork Thread 2 time 94ms
Trace 4 : EndWork Thread 2 time 104ms
Trace 5 : StartWork Thread 1 time 111ms
Trace 6 : StartWork Thread 2 time 112ms
Trace 7 : EndWork Thread 2 time 125ms
Trace 8 : EndWork Thread 1 time 127ms
```

**Potential problem here where Thread 2 has overtaken Thread 1, which may be unexpected**

# Debugging optimised code

- Disable instruction reordering
- Drop optimisation level
- Implement logging rather than single stepping code.

# Use asserts and capture core files

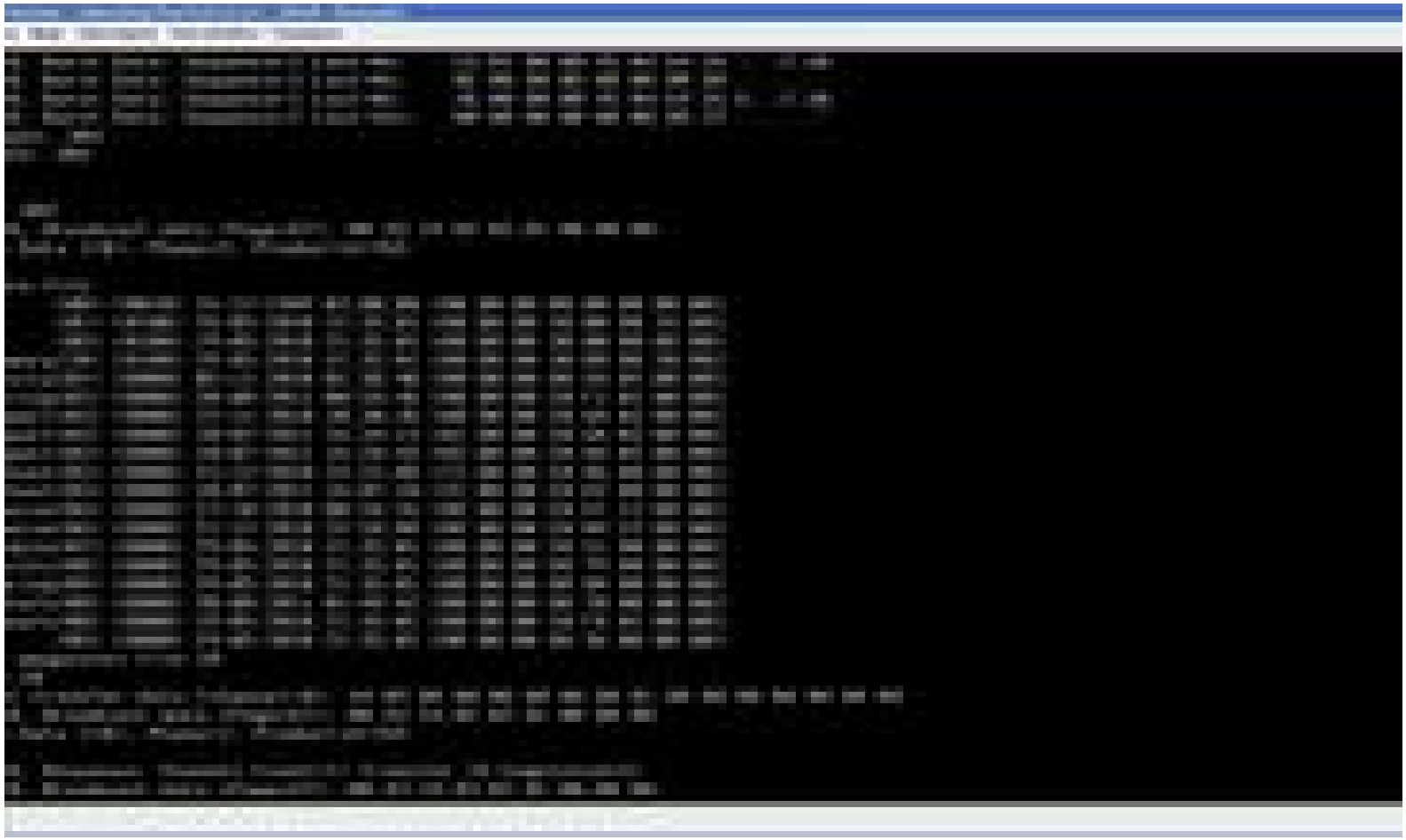
- Detect memory corruption by using asserts
- Captured core file can be run in the debugger to examine state

# Perform object hand-over rather than shared objects

- Manage object ownership explicitly rather than allowing all threads to access shared memory.
- Single thread owner and destroyer.

# Built-in debug support

- Produce your own application specific logging to assess state of failing applications after the fact



# RAII for mutex locking unlocking

- For functions that can throw assertions or have many exit points, utilise RAII for mutex locking and unlocking.

C++11 Example

```
int Thread1()
{
   MyLockingClass<std::mutex> myLock(gMutex);
   ...
} // MyLockingClass destructor is called here irrespective of
   how the function exited, releasing the mutex
```

C Example using GCC extension

```
int Thread()
{
   RAII_VARIABLE(MyMutex, mutex, lock(mutex), unlock());
   ...
}
```

# Conditional breakpoints/ ignore

- Only stop the process if a preset condition evaluates to true at the time the breakpoint is hit.
- Ignore the next X number of breakpoint fires before stopping

# Multi-threaded debugging shortcomings

- Single stepping multiple threads
- Viewing thread state
- Viewing data
- Reproducibility
- Architectural knowledge

# Single stepping multiple threads

- Stepping on one thread continues all other threads.
- Getting around this involves setting breakpoints everywhere
- This is difficult to manage and not easy to set up in the first place

# Viewing thread state

- How to represent thread state simultaneously.
- Requires to manually change active thread in gdb.
- Error prone if you step on the wrong thread.

# Viewing data

- How to view data on different threads simultaneously.
- Requires thread to be changed explicitly
- Difficult to visualise data for very large numbers of threads in this way

# Reproducibility

- How to reliably reproduce a failing state
- Capturing thread affinity
- Reapplying thread affinity

# Architectural knowledge

- Heap allocation
- Stack allocation
- Shared object data access between processes

# Things that need to be improved

- Single step/next will stop when any thread hits the next instruction / source line.
- Getting and setting the core affinity for threads spawned.
- Viewing thead local data across all threads.
- Automatic trace buffer insertion.

SAMSUNG
SRUK

# Thank you.