DAW Design & Implementation #4

Thread Synchronization & Lock Free Programming

Plan of Attack

- Basic overview of thread sync issues
- Standard solutions
- Problems
- Lock free introduction
- Ringbuffers
- Guard variables
- Read/Write Locks
- RCU

The Problems

- Data corruption/Race conditions
- Deadlocks

Data Corruption/Race Conditions

- Two threads operating on the same data
- OS offers no guarantees about ordering of thread execution (including interleaving)
- Thread A or B can be interrupted in the middle of its changes to the data
- Inconsistent views of the data
- No way to predict who wins

A Basic Race Condition

Thread1:

```
total = total + value1;
```

Thread2:

```
total = total + value2;
```

Under the hood

Each thread has to:

- Load the current value of `total` into a register
- Add the relevant value to that register
- Store the result back in the memory location for `total`

So that's 3 instructions

This is what we plan/want:

Thread1 load
Thread1 add
Thread1 store (total=total+value1)
Thread2 load
Thread2 add
Thread2 store (total=total+value1+value2)

This is what we could get

Thread1 load
Thread1 add
INTERRUPT + RESCHEDULE
Thread2 load
Thread2 add
Thread2 store (total=total+value2)
INTERRUPT + RESCHEDULE
Thread1 store (total=total+value1)

How to fix this?

- Use “locks” or “mutexes” to make sure that:
  - 1. only one thread is busy at a time
  - 2. certain operations appear atomic to everybody outside the thread that does them
- Locks are “acquired” and “held” & “released”
- Attempting to acquire a lock that is already held puts the calling thread to sleep
- Releasing a lock may or may not wake threads waiting to acquire it
- Wakeup order is not predictable

Thread 1 acquires lock
Thread1 load
Thread1 add
INTERRUPT + RESCHEDULE
Thread 2 tries to acquire lock, fails,
goes to sleep
INTERRUPT + RESCHEDULE
Thread1 store (total=total+value1)
Thread1 releases lock
INTERRUPT + RESCHEDULE
Thread2 wakes up, acquires lock
Thread2 load
Thread2 add
Thread2 store (total=total+value1+value2)
Thread2 releases lock

Deadlocks

- Caused by multiple different orderings of lock acquisition with > 1 lock

Thread1:
acquire lock A
acquire lock B

Thread2:
acquire lockB
acquire lockA

Thread1:
acquire lock A
INTERRUPT+ RESCHEDULE
acquire lock B

Thread2:
acquire lockB
INTERRUPT+RESCHEDULE
acquire lockA

Deadlocks are high level errors

- If an application uses more than 1 lock AND
- If more than 1 lock is held at any time THEN
- The application developer(s) MUST agree on the locking order AND
- It must NEVER be violated

General Rules

- Threads sharing data without mutual exclusion nearly always needs to race conditions
- Race conditions mean incorrect operation of the program, including but not necessarily implying crashing
- In your mind, imagine that your code can be interrupted ***ANYWHERE***

Rules for Realtime

- T = A(nframes) + B *where A and B are constant*
  - No calls to malloc() etc.
  - No disk I/O
  - No network I/O
  - No calls to sleep() etc.

The Problems with the Solution

- Realtime low-latency
- Our audio thread goes to acquire a lock protecting data shared with the GUI thread (e.g a message queue)
- The GUI holds the lock while it handles some previous message from the audio thread
- Audio thread blocks (sleeps) ... for how long?
- We cannot know
- Boom! Or rather “click”

Lock Timing

- Delay caused by blocking on a lock is unbounded
- Spinlock – don't go to sleep when you can't acquire the lock, just “spin” on the processor (e.g. adding to a temporary variable), and then try again
- Requires multiple processors
- Useful but requires a lot of knowledge about what threads are doing
- Best left to kernel developers

The Contradiction

- you must use locks
- you that you can't use locks (at least not in a realtime audio processing thread)

Lock Free Programming

- Been around for about 25 years
- Major motivation: improving OS performance on multiprocessor systems
- Secondary motivation: realtime usage
- General lock free methods: yawn
- Specific lock free methods: ok

Types of Lock-Free

- Two basic variants
- “Lock Free” - overall system (application) keeps moving forwards
- “Wait Free” - any given thread keeps moving forwards
- We need wait-free for RT work

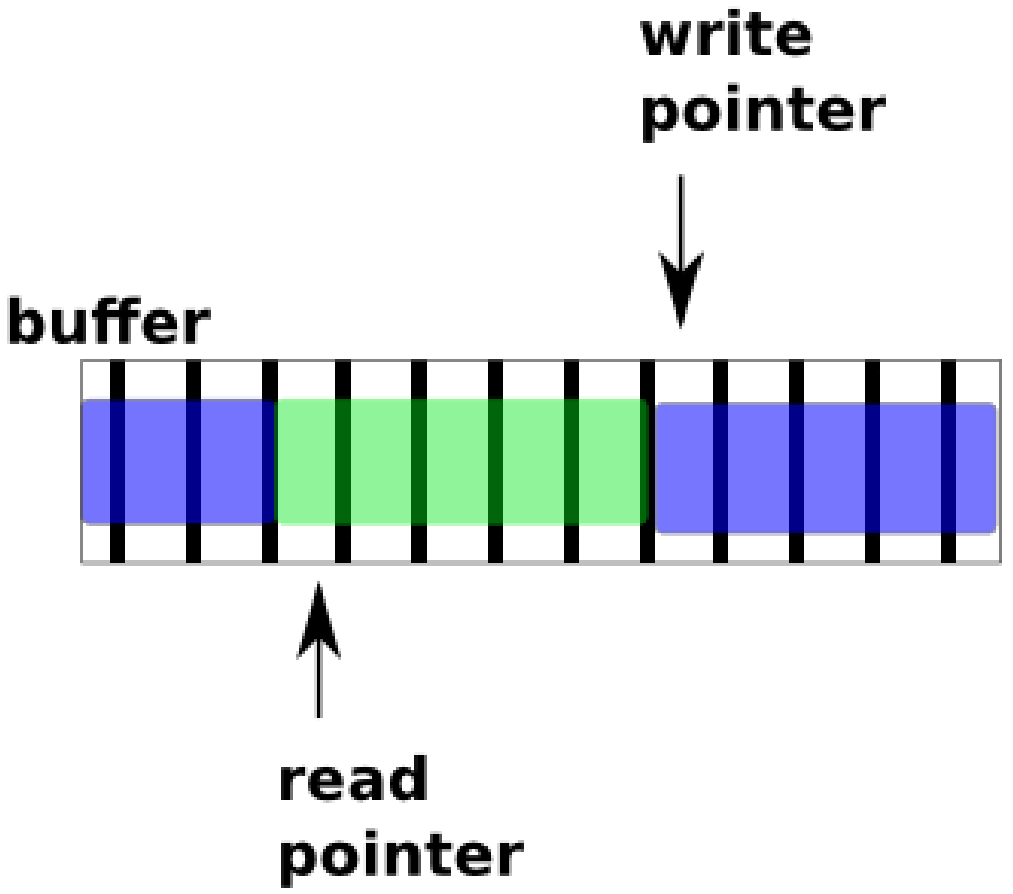
Lock Free Data Structures

- Use hardware primitives to compare and swap values atomically
- Useful for very specific cases
- Hard to use in general
- Still a bit of CS research problem

Ringbuffers

- Useful for passing data between two threads
- Could be “real” data, or “control”/“message” information
- No lock required for a specific case: 1 thread reads, 1 thread writes
- No exceptions
- Also known as a lock-free FIFO
- Provided for you by JACK, several implementations online

Ringbuffers: how they work



- Note that read/write space may need two pointers and sizes to describe it

Ringbuffer math

Read space =

```
if (w > r) { return w - r }
 else return (w - r + size) & size_mask;
```

- Assumes power of 2 size
- Can choose whether or not to wrap the “pointers” as they reach the end
- Wrapped: `w = (w + distance) & size_mask`
- Unwrapped: always use `w & size_mask` to access data

Ringbuffers: do they work?

- Later ...

Ringbuffers: what's in the buffer?

- Easy case: audio data – buffers contain sample values
- Medium-hard case – buffers contain pointers to messages, or messages? (malloc issues)
- Hard case – buffers contain closures
- C++ (libsigc++/boost) to the rescue

Guard Variables

- Singleton complex data structure
- Single writer, multiple readers
- Reading partially modified data structure would lead to errors
- Ideal where audio/RT thread is the writer, and “slow” thread(s) are the readers
- Otherwise decomposes to spinlock

How to use guard variables #1

```
Struct fooBar {
    int first;
    int second;
    float third;
    char fourth[32];
}
```

How to use guard variables #1

```
Struct fooBar {
    uint guard1;
    int first;
    int second;
    float third;
    char fourth[32];
    uint guard2;
}
```

Guard Variable writer:

```
theFooBar.guard1++;
... update theFooBar ...
theFooBar.guard2++;
```

Guard Variable Reader

```
FooBar val = theFooBar;
if (val.guard1 != val.guard2) {
      ... ??? ...
}  else {
    /* we got a consistent copy of fooBar in val */
}
```

About those ???

- guard1 != guard2
- What to do?
- Make another copy, and check that?
- What if that doesn't work?
- Sleep! We're not the RT thread
- Guard variables are about data consistency
- But ...

Memory Barriers

- Are there are any circumstances under which guard1 == guard2 but the variable update was still in progress?
- Yes!
- Thank your brilliant chip designer friends

Memory Barriers and Instruction Reordering (“out of order execution”)

- Modern processors can reorder their instruction streams for better performance

Old World CPU

1. Instruction fetch.

2. If input operands are available (in registers for instance), the instruction is dispatched to the appropriate functional unit. If one or more operands is unavailable during the current clock cycle (generally because they are being fetched from memory), however, the processor stalls until they are available.

3. The instruction is executed by the appropriate functional unit.

4. The functional unit writes the results back to the register file.

New World CPU

1. Instruction fetch.
2. Instruction dispatch to an instruction queue (also called instruction buffer or reservation stations).
3. The instruction waits in the queue until its input operands are available. The instruction is then allowed to leave the queue before earlier, older instructions.
4. The instruction is issued to the appropriate functional unit and executed by that unit.
5. The results are queued.
6. Only after all older instructions have their results written back to the register file, then this result is written back to the register file.

What were they thinking?

- avoid a processor stalling that occur when the data needed to perform an operation are unavailable
- Instead of stalling, execute other instructions that are ready
- Then re-order the results at the end to make it appear that the instructions were processed as normal
- Source code/assembler defines “program order”
- Data availability defines “data order”
- Actual execution order is “semi-random”

Real World

```
theFooBar.guard1++;
theFooBar.guard2++;
```

... update theFooBar ...

- Read sees consistent guard variables, but writer is not finished yet

Memory Barrier needed

```
theFooBar.guard1++;
  ... update theFooBar ...
hey_processor_make_sure_all_stores_are_done!
theFooBar.guard2++
```

- Memory barriers are very complicated, totally processor dependent, and only occasionally necessary ... such as ...
- OS Kernels
- Multithreaded applications using lock-free methods

Read/Write Locks

- Sometimes, you know a thread will not modify data, but just needs to read a consistent version of it
- Need to protect “reader” threads from activities of “writer” threads BUT
- No need to protect them from each other
- Declare 2 types of locks
- All readers must take the read lock – any number allowed
- All writers must take the write lock
- 1 writer blocks all readers
- 1 reader blocks all writers but not other readers

Read Copy Update

- This is cool
- Technically not lock-free
- Multiple (fast/RT) readers
- Slow writers
- Don't lock the data, copy the data
- Reader gets a copy to use, discards when done
- Writer gets a copy to use, swaps with original when done (“update”)
- Somebody has to clean up (slow writer is ideal)
- Absolutely reliant on `compare-and-swap` atomic operation
- Used for reference counting and the update step

RCU Example

- Data structure is a linked list of pointers to objects (e.g. tracks)
- RT thread needs to read the list for audio processing
- GUI thread could add/remove new objects/tracks from the list at any time
- Same idea for many data structures in a DAW (plugins, channels, ports and more)

A Template Example

- Use boost::shared_ptr<T> to provide reference-counted pointers – when they are no longer used, the objects they point to will be destroyed
- Use g_atomic_* for reading and CAS
- Two key methods:
- `reader()` - provides a “copy” for the readers
- `write_copy()` - provides a copy for the writers
- `update()` - accepts a copy from a writer and makes it the new value for everyone except any existing readers
- Note: the “value” is of type “pointer to shared_ptr to <T>”
- Smile!

What's in the RCU Manager?

- The basic data structure we are managing is `list<shared_ptr<Track*> >`
- The RCU Manager looks after `shared_ptr<list<shared_ptr<Track*> > >`
- Cleanup means that we delete the shared ptrs
- If these were the last references to a particular list (“copy”), the list will be deleted
- If that list contained the last reference to a Track object, then the Track object will be deleted

Consequences

- If a Track is in a list, it will not be deleted
- As long as a reader thread keeps its copy (a list) around, the Track will not go away
- The reader's copy will not be modified while in use (writer has its own copy)
- Thread safe
- Potentially lock free for reader
- Complex as hell!

Next Week

- Brief project presentation
- Parallel algorithms for audio applications
- http://tu.linuxaudiosystems.com/