# Lock Freedom

David Chisnall

March 8, 2011 (Pancake Day!)

# Locks are Slow!

- Cost to acquire and release
- System calls often required
- Can cause $n$ threads to block (wait) if a lock is accessible by $n + 1$ threads
- Possibility of deadlock
- Not ideal for high-performance computing!

# Wait Freedom

Every operation is bounded on the number of steps before completion.
(Never happens, back in the real world)

# Lock Freedom

- At least one thread must be able to make progress at any given time
- Eventually, all threads must make progress
- Given infinite time, infinitely many threads will progress

# Obstruction Freedom

A single thread, with all other threads paused, may complete its work.

# Implementing Obstruction Free Algorithms

- Requires strong guarantees on memory ordering
- Needs lots of thought!

# Problem 1: Compiler Reorders Memory Access

```
a = b;
b = c;
```

- Two store operations
- No dependencies
- Compiler is free to issue them in any order
- May also remove load operations if the value is already in a register!

# The volatile Keyword

```
volatile int a;
```

- The compiler must issue a memory read for every access to a
- The compiler must issue a memory write for every assignment to a
- The compiler may not re-order accesses and assignments to a
- The compiler is free to rearrange accesses to a relative to other memory access
- The compiler makes no guarantees about multithreaded access

# Problem 2: CPU Reorders Memory Access

- Most modern chips issue operations out of order
- Memory reads and writes may be reordered
- The processor will ensure that the current thread doesn't see the reordering...
- ...but other threads still can

# Memory Barriers

```
// GCC extension, full memory barrier:
__sync_synchronize();
```

- Provides a line in the instruction stream
- Memory accesses may not be reordered across the line
- Some architectures provide various forms of relaxed barriers (e.g. only writes may not be reordered)

# Example: Xen Time Source

- Hypervisor must provide guest VMs with current time
- Desire to avoid expensive calls from guest to hypervisor
- Lock-free mechanism for updating time

# Time in Xen

- Hypervisor provides coarse-grained time and time-stamp counter (TSC) value when it was accurate
- Generating the current time requires reading several values from memory
- What happens if your read overlaps with an update?

# Solution: Versioned Reads

```
struct shared_info
{
    int version, nanosecs, seconds, tscs;
};
struct shared_info atomic_read(volatile struct
   shared_info *info)
{
    struct ret;
    while ((ret->version = info->version) & 1) ;
    ret->nanosecs = info->nanosecs;
    ret->seconds = info->seconds;
    ret->tscs = info->seconds;
    if (ref->version == info->version)
        return ret;
    return atomic_read(info);
}
```

# Write Algorithm

```
info->version++;
__sync_synchronize();
info->nanosecs = nanosecs;
info->seconds = seconds;
info->tscs = seconds;
__sync_synchronize();
info->version++;
```

# Performance

Reader:

- No atomic operations required
- Common case just requires 5 reads
- Very fast!
- May need to retry if concurrent with write
- Unbounded worst-case time

Writer:

- Needs two barriers or atomic increments
- Similar cost to acquiring and releasing a mutex
- But never blocks - hard realtime guarantee for the writer!

# Example: Lockless Ring Buffer

- Producer-consumer problem
- Solution without locks
- Producer and consumer can both access queue concurrently!

# Simple Ring Buffer

1. Acquire lock
2. Insert object
3. Release lock

1. Acquire lock
2. Collect object
3. Release lock

How do we make this lock free?

# Potential Concurrency Problems

- Producer must find free space
- Consumer must find next item
- Producer must be able to tell if the buffer is full
- Consumer must be able to tell if the buffer is empty

# Solution: Free-running Counters

```
volatile uint32_t producer;
volatile uint32_t consumer;
int shift = 8;
// Must be power of two!
const bufferSize = 1<<shift;
const bufferMask = bufferSize - 1;
void *buffer[bufferSize];
```

# Inserting into the Ring

```
void insert(void *v)
{
    while (producer - consumer > bufferSize);
    buffer[producer & bufferMask] = v;
    producer++;
}
```

# Retrieving the Next Value

```
void* fetch(void)
{
    while (producer == consumer);
    void *v = buffer[consumer & bufferMask];
    consumer++;
    return v;
}
```

# Why it Works

- The counters are only ever read from one thread, written from the other
- Each update only increments a single shared variable
- Wrap-around is handled by overflow

# Better than Condition Variables?

- Very fast when the consumer and producer are both running
- Causes producer to spin when queue is full
- Causes consumer to spin when queue is empty
- Solution: Hybrid - use a condition variable only on boundary conditions (i.e. when transitioning to and from full / empty states)

# Questions?