Synchronising Threads

David Chisnall

March 1, 2011

First Rule for Maintainable Concurrent Code

No data may be both mutable and aliased

Harder Problems

- Data is shared and mutable
- Access to it must be protected

Mutual Exclusion

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
pthread_mutex_lock(&lock);
// Access protected data structure here
pthread_mutex_unlock(&lock);
```

- Mutual exclusion lock (mutex) can only be held by one thread
- All other threads block when they try to lock it

Atomic Access

- Often data structures need multiple operations to complete at once.
- Operations must appear as if they all happen at once to other threads

Simple Atomic Operations

Is this atomic?

```
variable++;
```

Simple Atomic Operations

It compiles to this:

```
movl    _variable, %eax
incl    %eax
movl    %eax, _variable
```

1. Load the value
2. Increment it
3. Store it

What happens if two threads try at once?

Simple Solution

```
pthread_mutex_lock(lock);
variable++;
pthread_mutex_unlock(lock);
```

The Problem with Locking

- All accesses to `variable` must be protected by the same mutex
- Mutex locking and unlocking is much more expensive than the operation we're doing!
- What happens for more complex operations?

Lock Order Problems

- Two data structures protected by locks
- What could possibly go wrong?
- If two threads try to acquire the locks in the opposite order, they deadlock

Simple Rules for Mutexes

- Hide data structure access in functions that handle the locking
- Hold locks for as little time as possible
- If you hold more than one lock, always acquire them in the same order
- If there are two possible solutions, and one involves using a mutex, use the other one

Atomic $\neq$ Thread Safe!

- Atomic operations are required for thread safety
- But thread safety requires other things
- Consider: given atomic set and get operations, how do you make atomic increment?

Read-Write Locks

- Common problem with shared data structures
- Any number of threads can read it at once
- Only one thread may modify it
- No thread may read the structure while it is being modified

Producer-Consumer Problem

- One thread produces data
- Another thread consumes it
- How do you synchronise the two?

Locks?

```
void producer(void)
{
    do
    {
        do_work();
        pthread_mutex_lock(&queueLock);
        add_data();
        pthread_mutex_unlock(&queueLock);
    } while(1);
}
```

Locks?

```
void consumer(void)
{
    do {
        data = NULL;
        while (NULL == data)
        {
            pthread_mutex_lock(&queueLock);
            data = get_work();
            pthread_mutex_unlock(&queueLock);
        }
        process_data();
    } while(1);
}
```

The Problem: Busy Waiting

- The consumer thread is constantly locking and unlocking the queue
- This consumes CPU time, even when it's not doing anything.
- This is especially bad on mobile devices (consumes battery)

Using Condition Variables

```
void producer(void)
{
    do
    {
        do_work();
        pthread_mutex_lock(&queueLock);
        add_data();
        // Wake up the other thread.
        pthread_cond_signal(&condVar);
        pthread_mutex_unlock(&queueLock);
    } while(1);
}
```

Using Condition Variables?

```
void consumer(void)
{
    do {
        data = NULL;
        pthread_mutex_lock(&queueLock);
        data = get_work();
        if (NULL == data)
        {
            pthread_cond_wait(&condVar,
                              &queueLock);
            data = get_work();
        }
        pthread_mutex_unlock(&queueLock);
        process_data();
    } while(1);
}
```

Using Condition Variables

- Must be protected by a mutex, or you can lose wakeup events
- Sleeping atomically releases the mutex, so the mutex must be owned first!
- Waking from sleep reacquires the mutex - don't forget to release it

Low-level Primitives

- All atomic operations are based on some atomic instructions
- x86 has a very rich set (atomic add, and so on)
- Most processors have one of two primitives

Atomic Compare-And-Swap

```
// Atomic version of this:
int cas(int *value, int old, int new)
{
if (*value == old)
{
*value = new;
return 1;
}
return 0;
}
```

1. Takes memory address, expected value, new value
2. Stores the new value if the old value is the expected one

Atomic Add with Compare-And-Swap

```
void atomic_increment(int *value)
{
    int old, new;
    do
    {
        old = value;
        new = old+1;
    } while (!cas(&value, a, a+1);
}
```

Keep trying until you succeed.

Exclusive Read / Write

- Special load / store instructions
- Must execute in pairs
- Store fails if address has been written to since load

Questions?