Section 1 / Atomics

Figure 1: Gurney Hallek

Threads

Suppose you run two copies of the same program at the same time. They both have a variable named `i`. Can a change to `i` made by one copy of the program impact the value of `i` in the other running copy of the program? Of course not.

Think of running two copies of a program at the same time as having two identical, but distinct, homes. What happens inside one house does not impact what happens inside the house next door.

Threads are a different way of getting more than one "copy" of a program to run at the same time. Threads are different, however, in that they all live within the same household. All of the housemates share the living space and all housemates have access to any global or shared resource. This makes for great gains in performance for a broad class of problems but also introduces great hazards.

Suppose you buy a carton of milk and place it in the fridge. If you were the only member of the household you would expect that when we next went to the fridge your milk would still be there, right? If you share the household with other people, this might not be the case.

Consider the following program:

```
#include <iostream>                                                   // 1
#include <thread>                                                     // 2
```

```cpp
#include <atomic>
#include <vector>

using std::cout;
using std::endl;
using std::atomic;
using std::vector;
using std::thread;

const uint32_t MAX_LOOPS = 10000;
const uint32_t NUM_THREADS = 16;

/*  volatile is necessary if any use of the optimizer
    is to be made.
*/
volatile uint32_t naked_int;
atomic<uint32_t> atomic_integer(0);

extern "C" void LoadLinkedStoreConditional(uint32_t * value);

void LLSCWorker() {
    extern volatile uint32_t naked_int;

    for (uint32_t i = 0; i < MAX_LOOPS; i++) {
        LoadLinkedStoreConditional((uint32_t *) &naked_int);
    }
}

void NakedWorker() {
    extern volatile uint32_t naked_int;

    for (uint32_t i = 0; i < MAX_LOOPS; i++) {
        naked_int++;
    }
}

void AtomicWorker() {
    extern atomic<uint32_t> atomic_integer;

    for (uint32_t i = 0; i < MAX_LOOPS; i++) {
        atomic_integer++;
    }
}

void DoNaked() {
    vector<thread *> threads;
    naked_int = 0;
    for (uint32_t i = 0; i < NUM_THREADS; i++) {
        threads.push_back(new thread(NakedWorker));
    }

    for (auto &t : threads) {
        t->join();
    }
}

void DoLLSC() {
    vector<thread *> threads;
    naked_int = 0;

    for (uint32_t i = 0; i < NUM_THREADS; i++) {
        threads.push_back(new thread(LLSCWorker));
    }

    for (auto &t : threads) {
        t->join();
    }
}

void DoAtomic() {
    vector<thread *> threads;

    for (uint32_t i = 0; i < NUM_THREADS; i++) {
        threads.push_back(new thread(AtomicWorker));
    }

    for (auto &t : threads) {
        t->join();
    }
}

int main() {

    DoNaked();
    DoAtomic();

    cout << "Correct sum is: ";
    cout << NUM_THREADS * MAX_LOOPS << endl;
    cout << "Naked sum: " << naked_int << endl;
    cout << "Atomic sum: " << atomic_integer << endl;

    DoLLSC();
    cout << "LLSC sum: " << naked_int << endl;
    return 0;
}
```

This program will spawn 16 threads which will each loop 10,000 times, adding one to a zero-initialized integer each loop. At the end, when all the threads complete, the integer should have the value 160,000.

Alas, this is an example of the class "Hidden Update" bug. The shared resource, the integer, will get clobbered in unpredictable ways.

For example, multiple runs might produce (snipped to show only the output from `NakedWorker()`):

- Naked sum: 74291
- Naked sum: 79390
- Naked sum: 89115
- etc

Not only are the results wrong, they are wrong in a different way each time.

Serializing Access to Integer Types

C++11 introduced the notion of *atomic integers*. These do not glow. Rather, access to them is guaranteed to be atomic… as in, cannot be broken down.

The hidden update problem's root cause is that adding (for example) to a value in memory involves three instructions at the assembly language level. A load, an addition, and a store. A hidden update occurs when a thread is yanked from the CPU in the middle of these instructions. When the thread returns to the CPU, the store causes its stale data to overwrite (hide) correct data.

There are many ways to avoid the hidden update problem including a large array of synchronization mechanisms. Alternatively, one can avoid the hidden update problem by ensuring the three instruction sequence isn't interrupted. This can be done using atomic integer types.

Using Atomics

First, make the appropriate include:

```
#include <atomic>
```

Next, make the appropriate declaration and initialize the variable. Here, replace "integral type" with some integer type:

```
atomic<integral type> atomic_integer(0);
```

Notice how the initial value is provided to the atomic variable's constructor.

Finally, use the atomic variable as you would any other integer.

A General Implementation

The newer ARM architectures provide a single instruction solution for addition, subtraction and various bitwise operations. These will be described below.

For ARMv8 and for later ARM versions (to perform operations other than those listed above), there is a general solution that isn't pretty. It is an example of Load Linked / Store Conditional. It isn't pretty because it involves a loop.

```
        .text                                                   // 1
        .p2align    2                                           // 2
                                                                // 3
#if defined(__APPLE__)                                          // 4
        .global     _LoadLinkedStoreConditional                 // 5
_LoadLinkedStoreConditional:                                    // 6
#else                                                           // 7
        .global     LoadLinkedStoreConditional                  // 8
LoadLinkedStoreConditional:                                     // 9
#endif                                                          // 10
1:      ldaxr       w1, [x0]                                    // 11
        add         w1, w1, 1                                   // 12
        stlxr       w2, w1, [x0]                                // 13
        cbnz        w2, 1b                                      // 14
        ret                                                     // 15
```

Lines 1 and 2 are boilerplate.

The conditional assembly block from line 4 through line 10 declare the label `LoadLinkedStoreConditional` as global for both Linux and Apple assemblers. The label itself is also stated.

It is worth explaining that labels marked as global must have an underscore prefix for Apple assembly.

This function is passed the address of an `int32_t`.

Line 11 loads the value found at that address into `w1` and also marks the address as needing watching (by the hardware).

Line 12 can be expanded and / or replaced with whatever operation needed to be done on the value.

Line 13 puts the potato on the fork. It is a store conditional which may or may not actually write anything to memory.

To understand this instruction, Kristien et al et al. provide this amazingly helpful picture:

The value returned in `w2` will be 0 of the store actually took place and will have the value of 1 if the store was rejected.

Imagine the following:

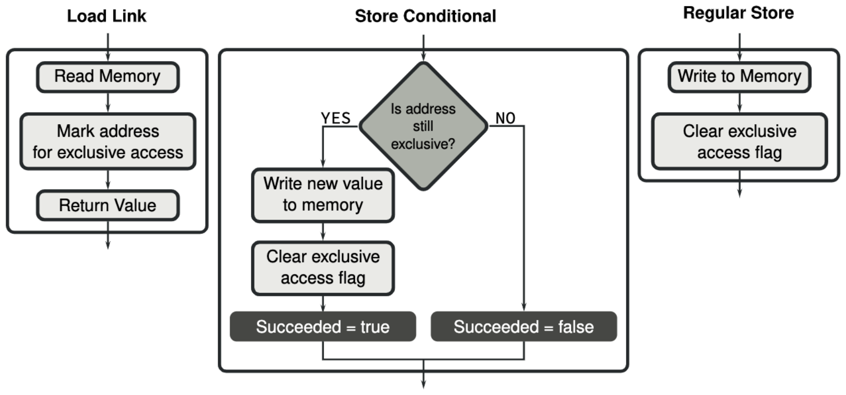
Load Link
Read Memory
Mark address for exclusive access
Return Value
Store Conditional
Is address still exclusive?
YES
NO
Write new value to memory
Clear exclusive access flag
Succeeded = true
Succeeded = false
Regular Store
Write to Memory
Clear exclusive access flag

Figure 2: llsc

T1	T2
Executes line 11. Gets value *N*. Location is marked. T1 is descheduled.	
	Executes line 11. Gets value *N*. Location is marked again. `w1` goes up to *N + 1* on line 12. Line 13 succeeds in storing *N + 1* to memory and the location marking is cleared T2 is descheduled.
T1 is scheduled - recall it has stale values. Executes line 12 making *N + 1* which is now wrong. Executes line 13 which fails because the marking is now gone. Loops around, this time picking up *N + 1* Correctly makes *N + 2*	

Implementation of ARMv8.1A and Newer

Implementations of operations on atomic variables were improved in the second version of ARMv8, called ARMv8.1. The load linked and store conditional instructions are still available but several new instructions were added which perform certain operations such as addition, subtraction and various bitwise operations in a single atomic instruction.

For example:

```
    mov       w1, 1
    ldaddal   w1, w0, [x0]
```

does the same work of atomically adding one to the value in memory pointed to by `x0`.