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>                                                    // 3
#include <vector>                                                    // 4
                                                                     // 5
using std::cout;                                                     // 6
using std::endl;                                                     // 7
using std::atomic;                                                   // 8
using std::vector;                                                   // 9
using std::thread;                                                   // 10
                                                                     // 11
const uint32_t MAX_LOOPS = 10000;                                    // 12
const uint32_t NUM_THREADS = 16;                                     // 13
                                                                     // 14
/*  volatile is necessary if any use of the optimizer                // 15
    is to be made.                                                   // 16
*/                                                                   // 17
volatile uint32_t naked_int = 0;                                     // 18
atomic<uint32_t> atomic_integer(0);                                  // 19
                                                                     // 20
void NakedWorker() {                                                 // 21
    extern volatile uint32_t naked_int;                              // 22
                                                                     // 23
    for (uint32_t i = 0; i < MAX_LOOPS; i++) {                       // 24
        naked_int++;                                                 // 25
    }                                                                // 26
}                                                                    // 27
                                                                     // 28
void AtomicWorker() {                                                // 29
    extern atomic<uint32_t> atomic_integer;                          // 30
                                                                     // 31
    for (uint32_t i = 0; i < MAX_LOOPS; i++) {                       // 32
        atomic_integer++;                                            // 33
    }                                                                // 34
}                                                                    // 35
                                                                     // 36
void DoNaked() {                                                     // 37
    vector<thread *> threads;                                        // 38
                                                                     // 39
    for (uint32_t i = 0; i < NUM_THREADS; i++) {                     // 40
        threads.push_back(new thread(NakedWorker));                  // 41
    }                                                                // 42
                                                                     // 43
    for (auto &t : threads) {                                        // 44
        t->join();                                                   // 45
    }                                                                // 46
}                                                                    // 47
                                                                     // 48
```

```
void DoAtomic() {                                                   // 49
    vector<thread *> threads;                                       // 50
                                                                    // 51
    for (uint32_t i = 0; i < NUM_THREADS; i++) {                    // 52
        threads.push_back(new thread(AtomicWorker));                // 53
    }                                                               // 54
                                                                    // 55
    for (auto &t : threads) {                                       // 56
        t->join();                                                  // 57
    }                                                               // 58
}                                                                   // 59
                                                                    // 60
int main() {                                                        // 61
                                                                    // 62
    DoNaked();                                                      // 63
    DoAtomic();                                                     // 64
                                                                    // 65
    cout << "Correct sum is: ";                                     // 66
    cout << NUM_THREADS * MAX_LOOPS << endl;                        // 67
    cout << "Naked sum: " << naked_int << endl;                     // 68
    cout << "Atomic sum: " << atomic_integer << endl;               // 69
                                                                    // 70
    return 0;                                                       // 71
}                                                                   // 72
perrykivolowitz@DAEDALUS atomics %
```

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:

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

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. The 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 old data to overwrite (hide) correct data.