When I memcpy a struct into a std::atomic of that struct, why does the result not match?

devblogs.microsoft.com/oldnewthing/20180328-00

March 28, 2018

Raymond Chen

Consider the following code:

```
// Code in italics is wrong.

struct Point3D { float x, y, z; };

std::atomic<Point3D> currentPoint;

bool LoadCurrentPointFromFile(HANDLE file)
{
 DWORD actualBytesRead;
 if (!ReadFile(file, &currentPoint, sizeof(Point3D),
               &actualBytesRead, nullptr)) return false;
 if (actualBytesRead != sizeof(Point3D)) return false;
 return true;
}
```

This code tries to load a `Point3D` structure from a file directly into a `std::atomic`. However, the customer found that the results were not properly loaded and suspected there may a bug in the `ReadFile` function, because the value that should have been in the `z` member ended up in `y`, the value that should have been in the `y` member ended up in `x`, and the value that should have been in the `x` member wasn't loaded at all.

The `ReadFile` function is working fine. What's wrong is that you aren't using the `std::atomic` variable properly.

The contents of a `std::atomic` variable are not directly accessible. You have to use methods like `store` and `load`. There are operator overloads which make atomic variables appear to be regular variables, but at no point can you get the address of the underlying `Point3D` storage.

Processors have restrictions on the sizes of operands on which they can natively perform atomic operations. Some restrictions apply to the size of the operand: Most processors do not support atomic operations on 12-byte objects, and it's not reasonable to expect a processor to be able to perform an atomic operation on a memory object that is megabytes in size, after all. Some restrictions are based on layout, such as whether the object is suitably aligned.

In the cases where the object cannot be managed atomically by the processor, the standard library steps in and adds a lock, and operations on the atomic variable take the lock to ensure that the operation is atomic. The reason everything is shifted is that the code took the address of the atomic variable itself, which includes the intenral lock, and the value you intended to read into `x` didn't vanish. It overwrote the lock!

Access to the contents of the atomic variable must be done by the appropriate methods on the atomic variable.

```
bool LoadCurrentPointFromFile(HANDLE file)
{
 DWORD actualBytesRead;
 Point3D point;
 if (!ReadFile(file, &point, sizeof(Point3D),
               &actualBytesRead, nullptr)) return false;
 if (actualBytesRead != sizeof(Point3D)) return false;
 currentPoint.store(point);
 return true;
}
```

There's a presentation from CppCon 2017 that covers `std::atomic` from start to finish, including performance characteristics. I'm going to consider this video to be homework, because next time I'm going to chatter about it.

Raymond Chen

Follow