Why does the compiler turn my conditional loop into an infinite one?

devblogs.microsoft.com/oldnewthing/20180924-00

September 24, 2018

Raymond Chen

A customer asked why the compiler turned their conditional loop into an infinite one.

```
#include <windows.h>

int x = 0, y = 1;
int* ptr;

DWORD CALLBACK ThreadProc(void*)
{
  Sleep(1000);
  ptr = &y;
  return 0;
}

int main(int, char**)
{
 ptr = &x; // starts out pointing to x

 DWORD id;
 HANDLE hThread = CreateThread(nullptr, 0, ThreadProc, 0, &id);

 // Wait for the thread to change the ptr
 // so that it points to a nonzero value
 while (*ptr == 0) { }

 return 0;
}
```

Translating into standard C++, for those who don't want to get bogged down in Windows-specific goop:

```
#include <chrono>
#include <thread>

int x = 0, y = 1;
int* ptr = &x;

void ThreadProc()
{
  std::this_thread::sleep_for(std::chrono::seconds(1));
  ptr = &y;
}

int main(int, char**)
{
 ptr = &x; // starts out pointing to x

 std::thread thread(ThreadProc);

 // Wait for the thread to change the ptr
 // so that it points to a nonzero value
 while (*ptr == 0) { }

 return 0;
}
```

The customer explained,

> The conditional loop becomes an infinite loop. The assembly code loads `ptr` into a register once (at the start of the loop), and then it compares the value pointed-to by that register against zero. It never reloads the `ptr` variable, so it never notices that the thread changed the value of `ptr` to point to a different value.
>
> We understand that if `ptr` is declared as `volatile int*`, then that will force the compiler to reload the `ptr` variable, which will then load to correct behavior.
>
> We'd like to understand why the compiler cannot be smart enough to turn off the optimization automatically. Clearly, this global variable will be accessed by more than one thread. So why can't the compiler do the right thing?

Okay, first the nitpick: The declaration `volatile int* ptr` does not make the `ptr` variable volatile. It defines `ptr` as a non-volatile pointer to a volatile integer. You wanted `int* volatile ptr`.

Back to the main question.

First: What's going on here?

Observe that in the loop, there are no accesses to `std::atomic` variables, nor are there any `std::memory_order` operations. This means that any changes to `ptr` or `*ptr` are a data race and consequently trigger undefined behavior.

(An intuitive way of thinking of this rule is "The compiler optimizes as if the program were single-threaded. The only points at which the compiler considers the possibility of multi-threading is when you access a `std::atomic` or apply a `std::memory_order`.")

That explains why the program doesn't behave as "expected". But what about the claim that the compiler should recognize this and disable the optimization?

Well, it struck me as odd to request that the compiler recognize that perhaps it's optimizing too much and intentionally "deoptimize" itself. And especially for the compiler to be able to look into the mind of the programmer and conclude, "Oh, this loop must be waiting for that global variable to change."

But suppose there's some rule in the compiler that says "If optimization results in an infinite loop, then go back and recompile the function with optimizations disabled." (Or maybe "keep turning off optimizations until you get something that isn't an infinite loop.") Aside from the surprise this rule might create, would that rule help?

Notice that in this case, we do not have an infinite loop. The loop will be broken if any thread does `x = 1` or `*ptr = 1`. It's not clear how much analysis the customer expects the compiler to do to scour the entire program to see if that is possible. Would it have to check every integer variable modification and try to see if that could possibly be a variable that `ptr` could point to?

Since it's not practical for the compiler to do a complete flow analysis to determine whether `x = 1` or `*ptr = 1` would ever occur, it would have to play it safe and assume it might.

Which means more generally that any access to global variables or references or pointers to data that could be shared between threads could not be cached because of the possibility that another thread modified the value between the two accesses.

```
int limit;

void do_something()
{
    ...
    if (value > limit)
        value = limit; // would have to re-fetch "limit"
    ...
    for (i = 0; i < 10; i++)
      array[i] = limit; // would have to re-fetch "limit"
    ...
}
```

You've basically declared open season on data races. "Go ahead and modify anything in any order from multiple threads. It's all good! Data races for you. Data races for you. Data races for everyone!"

But that's not the direction the C++ standard took. The C++ standard says that if you are going to modify a variable that is also being accessed by another thread, then you must use an atomic operation or enforce a memory order (which usually comes with a synchronization object).

So please do that.

Raymond Chen

Follow