The x86 architecture is the weirdo, part 2

devblogs.microsoft.com/oldnewthing/20220418-00

April 18, 2022

Raymond Chen

Some time ago I noted that The x86 architecture is the weirdo. (And by x86 I mean specifically x86-32.) I was reminded by the compiler folks of another significant place where the x86 architecture is different from all the others, and that's in how Windows structured exceptions are managed.

On Windows, all the other architectures track exception handling by using unwind codes and other information declared as metadata. If you step through a function on any other architecture, you won't see any instructions related to exception handling. Only when an exception occurs does the system look up the instruction pointer in the exception-handling information in the metadata, and use that to decide what to do: Which exception handler should run? What objects need to be destructed? That sort of thing.

But the x86 is the weirdo. On Windows, the x86 tracks exception information at runtime. When control enters a function that needs to deal with exceptions (either because it it wants to handle the exception, or just because it wants to run destructors when an exception is thrown out of the function), the code must create an entry in a linked list threaded through the stack and anchored by the value in `fs:[0]`. In the Microsoft Visual C++ implementation, the linked list node also contains an integer which represents the current progress through the function, and that integer is updated whenever there is a change to the list of objects requiring destruction. It is updated immediately after the construction of an object completes, and immediately before the destruction of an object commences.

This special integer is a real pain in the neck, because the optimizer sees it as a dead store and really wants to optimize it out. Indeed sometimes, it really *is* a dead store, but sometimes it isn't.

Consider:

```
struct S { S(); ~S(); };

void f1();
void f2();

S g()
{
    S s1;
    f1();
    S s2;
    f2();
    return S();
}
```

The code generation for this function goes like this:

```
struct ExceptionNode
{
    ExceptionNode* next;
    int (__stdcall *handler)(PEXCEPTION_POINTERS);
    int state;
};

S g()
{
    // Create a new node
    ExceptionNode node;
    node.next = fs:[0];
    node.handler = exception_handler_function;
    node.state = -1; // nothing needs to be destructed

    // Make it the new head of the linked list
    fs:[0] = &node;

    construct s1;
    node.state = 0; // s1 needs to be destructed

    f1();

    construct s2;
    node.state = 1; // s1 and s2 need to be destructed

    f2();

    construct return value;
    node.state = 2; // s1, s2, and return value need to be destructed

    node.state = 3; // s1 and return value need to be destructed
    destruct s2;

    node.state = 4; // return value needs to be destructed
    destruct s1;
}
```

The unwind state variable is updated whenever the list of "objects requiring destruction" changes. As far as the optimizer is concerned, all of these updates to `state` look like dead stores, since it seems that nobody reads them.

Aha, but somebody does read them: The `exception_handler_function`. The problem is that the call to the `exception_handler_function` is invisible: It is called when an exception is thrown by the `f1()` or `f2()` function, or by the destructor of the `S` objects.[1]

But wait, some of these really are dead stores. For example, the assignments of 2 to `node.state` is a dead store, because it is immediately followed by a store of 3, and there is nothing in between, so no exception could occur while the value is 2. Similarly, the store of 3

is dead because the destructor of `S` is implicitly `noexcept`.[1] And the store of 4 is dead for the same reason: No exception can occur when destructing `s1`.

Further dead store elimination becomes possible if `f1` or `f2` are changed to `noexcept`.

So the optimizer is in a tricky spot here: It wants to eliminate dead stores, but the simple algorithm for identifying dead stores doesn't work here because of the potential for exceptions.

Coroutines make this even worse: When a coroutine suspends, the exception-handling node needs to be copied from the stack into the coroutine frame, and then removed from the stack frame. And when the coroutine resumes, the state needs to be copied from the coroutine frame back into the stack, and linked into the chain of exception handlers.

Knowing exactly when to do this unlinking and relinking is tricky, because you still have to catch exceptions that occur in `await_suspend` and store them in the promise. But we learned that `await_suspend` is fragile because the coroutine may have resumed and run to completion before `await_suspend` returns.

```
void await_suspend(coroutine_handle<> handle)
{
  arrange_for_resumption(handle);
  throw oops; // who catches this?
}
```

The language says that the thrown exception is caught by the coroutine framework, which calls `promise.unhandled_exception()`. But the promise may no longer exist!

Dealing with all these crazy edge cases makes exception handling on x86, and particularly exception handling on x86 *in coroutines*, quite a complicated undertaking.

Bonus reading: Zero-cost exceptions aren't zero cost.

[1] Destructors default to noexcept if no members or base classes have potentially-throwing destructors, but you can mark your destructor as potentially-throwing,[2] and then exceptions thrown from destructors become something the compiler has to worry about.

[2] Please don't do that.

Raymond Chen

Follow