Why is the stack overflow exception raised before the stack has overflowed?

devblogs.microsoft.com/oldnewthing/20211217-00

December 17, 2021

Raymond Chen

Consider this program we looked at last time.

```
#include <stdio.h>

int maxdepth = 0;

int f()
{
  ++maxdepth;
  return f();
}

int main()
{
  __try {
    f();
  } __except (GetExceptionCode() == STATUS_STACK_OVERFLOW ?
              EXCEPTION_EXECUTE_HANDLER :
              EXCEPTION_CONTINUE_SEARCH) {
    printf("Overflow at %u\n", maxdepth);
  }
}
```

As before, make sure to compile this program with optimizations disabled to ensure that the recursive call occupies stack and doesn't get tail-call-optimized.

When you run this program, it will break into the debugger at the moment when the stack overflow occurs. And if you look at the stack pointer and the available stack space, you'll see that the stack overflow exception was raised before the stack actually overflowed.

The exception is raised while the stack still has around 12KB of space left. What's up with the extra 12KB? "I paid for that extra 12KB of memory, why won't you let me use it?"

The kernel grows the stack when the stack guard page exception is triggered. (We'll take a closer look at the stack guard page in a few months,) But when the kernel notices that the stack is nearly depleted, it raises the stack overflow exception. It does this *before* the stack is depleted so that there is still some stack on which to run the exception handlers.

The kernel is indeed letting you use that last 12KB of stack. It's letting you use it to handle the stack overflow exception!

In the above program, for example, there is a stack overflow handler that wants to print a message and then break out of the infinite recursion. If the kernel didn't raise the stack overflow exception until the stack was completely depleted, there wouldn't be any stack for the stack overflow handler to run on.

Now, you (the person who wants to extract every last drop out of your stack before it overflows) might say, "Well, the kernel could just allocate a special emergency stack for stack overflow exceptions."

But this would have to be a per-thread emergency stack, since multiple threads could be dealing with stack overflows simultaneously. And there's already a convenient chunk of per-thread data: The stack itself!

You could say that the kernel does indeed reserve some space for a per-thread emergency stack: It allocates it from the end of the stack. And the stack overflow exception is raised when you reach the emergency stack.

Bonus chatter: Switching to an emergency stack would create a lot of problems. It would mess up stack traces, since the stack trace at the point of a stack overflow would stop when it reached the end of the emergency stack. Split stacks aren't really a thing in Windows because the kernel needs to know the stack boundaries in order to detect stack buffer overflow attacks: If the chain of stack frames or the chain of exception records leaves the stack, then the kernel assumes that the stack is corrupted. This means that no exception handler from the main stack will be called when the system is running on the emergency stack. Switching to the emergency stack would be pointless, since there won't be any handlers to run anyway.

I guess the kernel *could* add extra support for an *emergency stack mode* where it knows that it's on an emergency stack and accept stack frames and exception handlers from the main stack. But that would be a lot of work compared to just carving the emergency stack out of the end of the regular stack. (It would also break any code that used `GetThreadStackLimits`.)

Raymond Chen

Follow