Determining approximately how much stack space is available, part 2

devblogs.microsoft.com/oldnewthing/20200610-00

June 10, 2020

Raymond Chen

Last time, we used the `_alloca` function as a sneaky way to probe whether there was sufficient stack space available. However, it had a number of downsides, as previously discussed.

Another way to determine whether there is sufficient stack space is to calculate it yourself. The `GetCurrentThreadStackLimits` returns the current stack bounds. You can compare the results against the current stack pointer to see how much room is left.

```
__declspec(noinline)
bool is_stack_available(size_t amount)
{
  ULONG_PTR low, high;
  GetCurrentThreadStackLimits(&low, &high);
  auto remaining = reinterpret_cast<ULONG_PTR>(&low) - low;
  if (remaining > high - low) {
    __fastfail(FAST_FAIL_INCORRECT_STACK);
  }
  return remaining >= amount;
}
```

This function obtains the current stack pointer by taking the address of a local variable. This is only an approximation, because the compiler could choose to put the local variable in unused parameter home space or in the red zone, but those locations are close to the stack pointer, so it's basically good enough.

The distance from the current stack pointer to the bottom of the stack is the total remaining stack space. This assumes that the stack grows downward, but that's true of every Win32 processor, so we're okay there, for now.[1]

As a sanity check, we validate that the calculated value for remaining stack is reasonable. If the stack pointer were discovered to be below the low limit or above the high limit, then we fail fast declaring that the stack is corrupt. (We can do this with a single comparison thanks to the required wrapping behavior of unsigned arithmetic.)

Note that this mechanism differs from the `_alloca` technique in a few ways. One is that it does not commit any of the pages in the remaining stack; it merely reports how much is possible without forcing it to become realized. This is a good thing, because it means that you don't end up pre-paying for something you may not actually need.

Another difference is that that it does not take the thread stack guarantee into account. The thread stack guarantee sets the point at which the system raises a stack overflow exception. The memory reserved by the guarantee is therefore not available for general use. Therefore, this version will over-report available stack. We can take the guarantee into account with a little more tweaking:

```
__declspec(noinline)
bool is_stack_available(size_t amount)
{
  ULONG_PTR low, high;
  GetCurrentThreadStackLimits(&low, &high);
  auto remaining = reinterpret_cast<ULONG_PTR>(&low) - low;
  if (remaining > high - low) {
    __failfast(FAST_FAIL_INCORRECT_STACK);
  }
  ULONG guarantee = 0;
  SetThreadStackGuarantee(&guarantee);
  return remaining >= amount + guarantee;
}
```

We call `SetThreadStackGuarantee` with a guarantee of zero to query the current guarantee. We then report whether the remaining size is enough to cover the requested size plus the guarantee.

The `noinline` attribute is important for a subtle reason: Without it, you may find false positive fail-fast exceptions. Can you figure out why?

The answer is coroutines.

If the function is inlined into a coroutine, then the "stack" variables are more likely to be stored in the coroutine frame on the heap. Compiler optimizations may be able to figure out that a local variable's lifetime does not cross a suspension point, so it can be moved to the stack, but that is at the compiler's discretion and not guaranteed. Marking the function as `noinline` prevents it from being inlined into a coroutine.[2]

[1] Pour one out for the poor Itanium.

[2] Our previous version with `_alloca` also had this problem if inlined into a coroutine. In that case, we are saved because `_alloca` is disallowed in a coroutine, so the opportunity never arises.

Raymond Chen

Follow