The SuspendThread function suspends a thread, but it does so asynchronously

devblogs.microsoft.com/oldnewthing/20150205-00

February 5, 2015

Raymond Chen

Prologue: Why you should never suspend a thread.

Okay, so a colleague decided to ignore that advice because he was running some experiments with thread safety and interlocked operations, and suspending a thread was a convenient way to open up race windows.

While running these experiments, he observed some strange behavior.

```
LONG lValue;

DWORD CALLBACK IncrementerThread(void *)
{
 while (1) {
  InterlockedIncrement(&lValue);
 }
 return 0;
}

// This is just a test app, so we will abort() if anything
// happens we don't like.

int __cdecl main(int, char **)
{
 DWORD id;
 HANDLE thread = CreateThread(NULL, 0, IncrementerThread, NULL, 0, &id);
 if (thread == NULL) abort();

 while (1) {
  if (SuspendThread(thread) == (DWORD)-1) abort();

  if (InterlockedOr(&lValue, 0) != InterlockedOr(&lValue, 0)) {
   printf("Huh? The variable lValue was modified by a suspended thread?\n");
  }

  ResumeThread(thread);
 }
 return 0;
}
```

The strange thing is that the "Huh?" message was being printed. How can a suspended thread modify a variable? Is there some way that `InterlockedIncrement` can start incrementing a variable, then get suspended, and somehow finish the increment later?

The answer is simpler than that. The `SuspendThread` function tells the scheduler to suspend the thread but does not wait for an acknowledgment from the scheduler that the suspension has actually occurred. This is sort of alluded to in the documentation for `SuspendThread` which says

> This function is primarily designed for use by debuggers. It is not intended to be used for thread synchronization

You are not supposed to use `SuspendThread` to synchronize two threads because there is no actual synchronization guarantee. What is happening is that the `SuspendThread` signals the scheduler to suspend the thread and returns immediately. If the scheduler is busy doing something else, it may not be able to handle the suspend request immediately, so the thread being suspended gets to run on borrowed time until the scheduler gets around to processing the suspend request, at which point it actually gets suspended.

If you want to make sure the thread really is suspended, you need to perform a synchronous operation that is dependent on the fact that the thread is suspended. This forces the suspend request to be processed since it is a prerequisite for your operation, and since your operation is synchronous, you know that by the time it returns, the suspend has definitely occurred.

The traditional way of doing this is to call `GetThreadContext`, since this requires the kernel to read from the context of the suspended thread, which has as a prerequisite that the context be saved in the first place, which has as a prerequisite that the thread be suspended.

Raymond Chen

Follow