If I signal an auto-reset event and there is a thread waiting on it, is it guaranteed that the event will be reset and the waiting thread released before SetEvent returns?

devblogs.microsoft.com/oldnewthing/20130816-00

August 16, 2013

Raymond Chen

Let's go straight to the question:

> I have two programs that take turns doing something. Right now, I manage the hand-off with two auto-reset events. In Thread A, after it finishes doing some work, it signals Event B and then immediately waits on Event A. Thread B does the converse: When its wait on Event B completes, it does some work, then signals Event A and then immediately waits on Event B.
>
> This works great, but I'm wondering if I can save myself an event and use the same event to hand control back and forth. Is it guaranteed that when Thread A signals Event B, that this will release Thread B and reset the event (since it is auto-reset) before the call to `SetEvent` returns? If so, then I can just have one event and use it to bounce control back and forth.

Let's try it!

```
#include <windows.h>
#include <stdio.h>
HANDLE h;
DWORD CALLBACK ThreadProc(void *msg)
{
 for (;;) {
  SetEvent(h);
  // The theory is that the above SetEvent does not return
  // until the other thread has positively completed its wait,
  // so this upcoming wait will not complete until the other
  // thread calls SetEvent.
  WaitForSingleObject(h, INFINITE);
  puts((LPSTR)msg);
 }
}
int __cdecl main(int, char**)
{
 DWORD id;
 h = CreateEvent(0, FALSE, TRUE, 0);
 CloseHandle(CreateThread(0, 0, ThreadProc, "T1", 0, &id));
 CloseHandle(CreateThread(0, 0, ThreadProc, "T2", 0, &id));
 Sleep(INFINITE);
 return 0;
}
```

If you run this program, you'll see that the two threads come nowhere near taking turns. Instead, you see stretches where thread T1 gets to run a whole bunch of iterations in a row, and stretches where thread T2 gets to run a whole bunch of iterations in a row.

Okay, so we have demonstrated by experiment that this technique does not work. (You can use experimentation to show that something doesn't always work, but you can't use it to show that something always *will* work. For that you need to read some contracts and put on your thinking cap.) But why doesn't it work?

The lawyerly explanation for why it doesn't work is that there is nothing in the contract that says that it *does* work. Perfectly correct, but not particularly insightful.

Signaling an event makes all waiting threads eligible to run, but that doesn't mean that they actually *will* run. One of the waiting threads is woken to say "Hey, now's your chance." But that thread might be groggy and slow to wake, and in the meantime, another thread can swoop in and steal the event signal. And then that groggy thread shuffles downstairs to the breakfast table to find that somebody ate his pancake. (Actually, in principle, the kernel could just make it a total free-for-all and wake *all* the waiting threads, but I suspect it just picks one.)

We saw earlier that the thread that you would expect to run next might be temporarily unavailable and miss its chance to claim what it thinks is rightfully his. And more recent versions of Windows have exacerbated the problem by abandoning fairness in order to

improve throughput and avoid lock convoys. Now, in principle, the kernel could have reset the event when it woke the waiting thread, thereby assigning the wake to the thread at signal time, but that would have reintroduced the problem that unfairness was trying to solve.

The irony here is that what you're doing here is intentionally trying to *create* a convoy, and you're running into the scheduler's convoy-resistance.

Just use the two-event pattern. That makes it explicit what you want.

Raymond Chen

Follow