Consequences of the scheduling algorithm: Low priority threads can run even when higher priority threads are running

devblogs.microsoft.com/oldnewthing/20051003-08

October 3, 2005

Raymond Chen

Just because you have a thread running at a higher priority level doesn't mean that no threads of lower priority will ever run.

Occasionally, I see people write multi-threaded code and put one thread's priority higher than the other, assuming that this will prevent the lower-priority thread from interfering with the operation of the higher-priority thread so that they don't need to do any explicit synchronization.

```
BOOL g_fReady;
int g_iResult;
// high priority thread
SetResult(int iResult)
{
 g_fReady = TRUE;
 g_iResult = iResult;
}


// low priority thread
if (g_fReady)
{
 UseResult(g_iResult);
}
```

Let's ignore the cache coherency elephant in the room. If there were a guarantee that the low priority thread will never ever run while the high priority thread is running, this code looks okay. Even if the high priority thread interrupts and sets the result after the low priority thread has checked the ready flag, all that happens is that the low priority thread misses out on the result. (This is hardly a new issue, since the principle of relativity of simultaneity says that this was a possibility anyway.)

However, there is no guarantee that the low priority thread can't interfere with the high priority thread.

The scheduler's rule is to look for the thread with the highest priority that is "runnable", i.e., ready to run, and assign it to a CPU for execution. To be ready to run, a thread cannot be blocked on anything, and it can't already be running on another CPU. If there is a tie among runnable threads for the highest priority, then the scheduler shares the CPU among them roughly equally.

You might think that, given these rules, as long as there is a high priority thread that is runnable, then no lower-priority thread will run. But that's not true.

Consider the case of a multi-processor system (and with the advent of hyperthreading, this is becoming more and more prevalent), where there are two runnable threads, one with higher priority than the other. The scheduler will first assign the high-priority thread to one of the processors. But it still has a spare CPU to burn, so the low-priority thread will be assigned to the second CPU. You now have a lower priority thread running simultaneously as a higher priority thread.

Of course, another way a lower priority thread can run even though there are higher priority threads in the system is simply that all the higher priority threads are blocked. In addition to the cases you might expect, namely waiting on a synchronization object such as a semaphore or a critical section, a thread can also block for I/O or for paging. Paging is the wildcard here, since you don't have any control over when the system might decide to page out the memory you were using due to memory pressure elsewhere in the system.

The moral of the story is that thread priorities are not a substitute for proper synchronization.

Next time, a reversal of this fallacy.

Raymond Chen

Follow