Polling by sleeping versus polling by waiting with a timeout

devblogs.microsoft.com/oldnewthing/20090727-00

July 27, 2009

Raymond Chen

Commenter Francois Boucher asks it's better to write a background worker thread that polls with `Sleep()` and a flag, or polls by waiting for an event with a timeout?

```
// method A
while (fKeepGoing) {
 .. a little work ..
 Sleep(50);
}
// method B
do {
 .. a little work ..
} while (WaitForSingleObject(hEvent, 50) == WAIT_TIMEOUT);
```

"Which scenario is better? The first one uses only 1 handle for the thread. The second one will use 2. But is the first scenario wasting more thread time? Is it worth using the event (kernel object)?"

Yeah, whatever.

I don't quite understand why you want to pause every so often. Why not just do the work at low priority? When there are more important things to do, your background thread will stop doing whatever it's doing. When there is an available CPU, then your background thread will do its thing as fast as it can (until something with higher priority arrives).

The only thing I can think of is that by adding the pauses, your program won't look like it's consuming 100% CPU while the background processing is going on. Which is true, but then again, that's not much consolation. "Wow, with these changes, my spell check takes only 10% of the CPU. But then again, it also takes 10 times as long." Is that an improvement? You made your customer wait ten times as long for the document to be spell checked. That's ten times less battery life for your laptop.

And generally speaking, polling should be avoided because it carries negative consequences for system performance. So we're basically asking, "Which is better, hammering with a screwdriver or hammering with pliers?"

But let's say for the sake of argument that this "back off periodically" polling loop is actually the right design, and the only argument is which of the above two methods is "better" in terms of the criteria listed above (handles, thread time).

It still doesn't matter.

Method A has one fewer handle, but one more flag. So the total number of things you have to keep track of is the same either way.

"Oh, but I save the overhead of an additional handle."

Dude, you're already burning a thread. A single handle to an event is noise compared to the cost of a thread.

"But I save the cost of validating the handle and following the reference to the underlying kernel object."

Dude, you're about to go to sleep for 50 *milliseconds*. Saving a few thousand clocks is noise compared to 50 milliseconds.

The flag method does have some other problems, none of which are deal-breaking, but are things to bear in mind.

First, that flag that you're checking. There's no synchronization on that variable, so the background thread may run a smidge longer than necessary because the change hasn't yet propagated to the CPU running the loop. Similarly, the sleep loop does take a little longer to notice that it should stop working. If the `fKeepGoing` flag is set to `FALSE` during the sleep, the sleep will still run to completion before the loop finds out.

In the grand scheme of things, however, the extra 50 to 100 milliseconds are probably not a big deal. The background thread is a little slow to shut down, that's all. The user will probably not even notice that the CPU meter was higher than normal for an additional tenth of a second. After all, the typical human reaction time is 100 milliseconds anyway.

I'm assuming that the code that signals the background thread doesn't sit around waiting for the background thread to clean up. If it does, then this 100ms delay may start to combine with other delays to turn into something the user may start to notice. A tenth of a second here, a tenth of a second there, soon you may find yourself accumulating a half second's delay, and that's a delay the human brain can perceive.