Understanding the consequences of WAIT_ABANDONED

devblogs.microsoft.com/oldnewthing/20050912-14

September 12, 2005

Raymond Chen

One of the important distinctions between mutexes and the other synchronization objects is that mutexes have owners. If the thread that owns a mutex exits without releasing the mutex, the mutex is automatically released on the thread's behalf.

But if this happens, you're in big trouble.

One thing many people gloss over is the `WAIT_ABANDONED` return value from the synchronization functions such as `WaitForSingleObject`. They typically treat this as a successful wait, because it does mean that the object was obtained, but it also tells you that the previous owner left the mutex abandoned and that the system had to release it on the owner's behalf.

Why are you in big trouble when this happens?

Presumably, you created that mutex to protect multiple threads from accessing a shared object while it is an unstable state. Code enters the mutex, then starts manipulating the object, temporarily making it unstable, but eventually restabilizing it and then releasing the mutex so that the next person can access the object.

For example, you might have code that manages an anchored doubly-linked list in shared memory that goes like this:

```
void MyClass::ReverseList()
{
 WaitForSingleObject(hMutex, INFINITE);
 int i = 0; // anchor
 do {
  int next = m_items[i].m_next;
  m_items[i].m_next = m_items[i].m_prev;
  m_items[i].m_prev = next;
  i = next;
 } while (i != 0);
 ReleaseMutex(hMutex);
}
```

There is nothing particularly exciting going on. Basic stuff, right?

But what if the program crashes while holding the mutex? (If you believe that your programs are bug-free, consider the possiblity that the program is running over the network and the network goes down, leading to an in-page exception. Or simply that the user went to Task Manager and terminated your program while this function is running.)

In that case, the mutex is automatically released by the operating system, leaving the linked list in a corrupted state. The next program to claim the mutex will receive `WAIT_ABANDONED` as the status code. If you ignore that status code, you end up operating on a corrupted linked list. Depending on how that linked list is used, it may result in a resource leak or the system creating an unintended second copy of something, or perhaps even a crash. The unfortunate demise of one program causes other programs to start behaving strangely.

Then again, the question remains, “What do you do, then, if you get `WAIT_ABANDONED`?” The answer is, “Good question.”

You might try to repair the corruption, if you keep enough auxiliary information around to recover a consistent state. You might even design your data structures to be transactional, so that the death of a thread manipulating the data structures does not leave them in a corrupted state. Or you might just decide that since things are corrupted, you should throw away everything and start over, losing the state of work in progress, but at least allowing new work to proceed unhindered.

Or you might simply choose to ignore the error and continue onwards with a corrupt data structure, hoping that whatever went wrong won’t result in cascade failures down the line. This is what most people do, though usually without even being aware that they’re doing it. And it’s really hard to debug the crashes that result from this approach.

Exercise: Why did we use indices instead of pointers in our linked list data structure?

[Raymond is currently away; this message was pre-recorded.]

Raymond Chen

Follow