How do I determine whether I own a critical section if I am not supposed to look at internal fields?

devblogs.microsoft.com/oldnewthing/20031222-00

December 22, 2003

Raymond Chen

Seth asks how he can perform proper exception-handling cleanup if he doesn't know whether he needs to clean up a critical section.

> I'm using SEH, and have some __try/__except blocks in which the code enters and leaves critical sections. If an exception occurs, I don't know if I'm currently in the CS or not. Even wrapping the code in __try/__finally wouldn't solve my problems.

Answer: You know whether you own the CRITICAL_SECTION because you entered it.

Method 1: Deduce it from your instruction pointer.
"If I'm at this line of code, then I must be in the critical section."

```
__try {
  ...
  EnterCriticalSection(x);
  __try { // if an exception happens here
     ...  // make sure we leave the CS
  } __finally { LeaveCriticalSection(x); }
  ...
} except (filter) {
  ...
}
```

Note that this technique is robust to nested calls to EnterCriticalSection. If you take the critical section again, then wrap the nested call in its own try/finally.

Method 2: Deduce it from local state
"I'll remember whether I entered the critical section."

```
int cEntered = 0;
__try {
  ...
  EnterCriticalSection(x);
  cEntered++;
  ...
  cEntered--;
  LeaveCriticalSection(x);
  ...
} except (filter) {
  while (cEntered--)
    LeaveCriticalSection(x);
  ...
}
```

Note that this technique is also robust to nested calls to EnterCriticalSection. If you take the critical section again, increment cEntered another time.

Method 3: Track it in an object

Wrap the CRITICAL_SECTION in another object.
This most closely matches what Seth is doing today.

```
class CritSec : CRITICAL_SECTION
{
public:
 CritSec() : m_dwDepth(0), m_dwOwner(0)
   { InitializeCriticalSection(this); }
 ~CritSec() { DeleteCriticalSection(this); }
 void Enter() { EnterCriticalSection(this);
    m_dwDepth++;
    m_dwOwner = GetCurrentThreadId(); }
 void Leave() { if (!--m_dwDepth) m_dwOwner=0;
    LeaveCriticalSection(this); }
 bool Owned()
   { return GetCurrentThreadId() == m_dwOwner; }
private:
  DWORD m_dwOwner;
  DWORD m_dwDepth;
};
__try {
  assert(!cs.Owned());
  ...
  cs.Enter();
  ...
  cs.Leave();
  ...
} except (filter) {
  if (cs.Owned()) cs.Leave();
}
```

Notice that this code is **not** robust to nested critical sections (and correspondingly, Seth's code isn't either). If you take the critical section twice, the exception handler will only leave it once.

Note also that we assert that the critical section is not initially owned. If it happens to be owned already, then our cleanup code may attempt to leave a critical section that it did not enter. (Imagine if an exception occurs during the first "…".)

Method 4: Track it in a smarter object
Wrap the CRITICAL_SECTION in a smarter object.
Add the following method to the CritSec object above:

```
 DWORD Depth() { return Owned() ? m_dwDepth : 0; }
```

Now you can be robust to nested critical sections:

```
DWORD dwDepth = cs.Depth();
__try {
  ...
  cs.Enter();
  ...
  cs.Leave();
  ...
} except (filter) {
  while (cs.Depth() > dwDepth)
    cs.Leave();
}
```

Note however that I am dubious of the entire endeavor that inspired the original question.

Cleaning up behind an exception thrown from within a critical section raises the issue of "How do you know what is safe to clean up?" You have a critical section because you are about to destabilize a data structure and you don't want others to see the data structure while it is unstable. But if you take an exception while owning the critical section – well your data structures are unstable at the point of the exception. Merely leaving the critical section will now leave your data structures in an inconsistent state, leading to harder-to-diagnose bugs later on. "How did my count get out of sync?"

More rants on exceptions in a future entry.

Exercise: Why don't we need to use synchronization to protect the uses of m_dwDepth and m_dwOwner?

Update 2004/Jan/16: Seth pointed out that I got the two branches of the ternary operator backwards in the Depth() function. Fixed.

Raymond Chen

Follow