# If you protect a write with a critical section, you may also want to protect the read

devblogs.microsoft.com/oldnewthing/20111130-00

November 30, 2011

Raymond Chen

It is common to have a critical section which protects against concurrent writes to a variable or collection of variables. But if you protect a write with a critical section, you may also want to protect the read, because the read might race against the write.

Adam Rosenfield shared his experience with this issue in a comment from a few years back. I'll reproduce the example here in part to save you the trouble of clicking, but also to make this entry look longer and consequently make it seem like I'm actually doing some work (when in fact Adam did nearly all of the work):

```
class X {
 volatile int mState;
 CRITICAL_SECTION mCrit;
 HANDLE mEvent;
};
X::Wait() {
 while(mState != kDone) {
   WaitForSingleObject(mEvent, INFINITE);
 }
}
X::~X() {
 DestroyCriticalSection(&mCrit);
}
X::SetState(int state) {
 EnterCriticalSection(&mCrit);
 // do some state logic
 mState = state;
 SetEvent(mEvent);
 LeaveCriticalSection(&mCrit);
}
Thread1()
{
 X x;
 ... spawn off thread 2 ...
 x.Wait();
}
Thread2(X* px)
{
 ...
 px->SetState(kDone);
 ...
}
```

There is a race condition here:

- Thread 1 calls `X::Wait` and waits.
- Thread 2 calls `X::SetState`.
- Thread 2 gets pre-empted immediately after calling `SetEvent`.
- Thread 1 wakes up from the `WaitForSingleObject` call, notices that `mState == kDone`, and therefore returns from the `X::Wait` method.
- Thread 1 then destructs the `X` object, which destroys the critical section.
- Thread 2 finally runs and tries to leave a critical section that has been destroyed.

The fix was to enclose the *read* of `mState` inside a critical section:

```
X::Wait() {
 while(1) {
   EnterCriticalSection(&mCrit);
   int state = mState;
   LeaveCriticalSection(&mCrit);
   if(state == kDone)
     break;
   WaitForSingleObject(mEvent, INFINITE);
 }
}
```

Forgetting to enclose the read inside a critical section is a common oversight. I've made it myself more than once. You say to yourself, "I don't need a critical section here. I'm just reading a value which can safely be read atomically." But you forget that the critical section isn't just for protecting the write to the variable; it's also to protect all the other actions that take place under the critical section.

And just to make it so I actually did some work today, I leave you with this puzzle based on an actual customer problem:

```
class BufferPool {
public:
 BufferPool() { ... }
 ~BufferPool() { ... }
 Buffer *GetBuffer()
 {
  Buffer *pBuffer = FindFreeBuffer();
  if (pBuffer) {
   pBuffer->mIsFree = false;
  }
  return pBuffer;
 }
 void ReturnBuffer(Buffer *pBuffer)
 {
  pBuffer->mIsFree = true;
 }
private:
 Buffer *FindFreeBuffer()
 {
  EnterCriticalSection(&mCrit);
  Buffer *pBuffer = NULL;
  for (int i = 0; i < 8; i++) {
   if (mBuffers[i].mIsFree) {
    pBuffer = &mBuffers[i];
    break;
   }
  }
  LeaveCriticalSection(&mCrit);
  return pBuffer;
 }
private:
 CRITICAL_SECTION mCrit;
 Buffer mBuffers[8];
};
```

The real class was significantly more complicated than this, but I've distilled the problem to its essence.

The customer added, "I tried declaring `mIsFree` as a volatile variable, but that didn't seem to help."

Raymond Chen

**Follow**