Even if you have a lock, you can borrow some lock-free techniques

devblogs.microsoft.com/oldnewthing/20110422-00

April 22, 2011

Raymond Chen

Even if you prefer to use a lock (after all, they are much easier to program), you can borrow some lock-free techniques. For example, consider this:

```
CRITICAL_SECTION g_cs;
GORILLADATA g_data;
void PokeGorilla(double intensity)
{
  EnterCriticalSection(&g_cs);
  DeformGorilla(intensity, &g_data);
  Reticulate(&g_data.spline);
  int stress = CalculateTension(&g_data.spline);
  if (stress < 25)      g_data.mood = RELAXED;
  else if (stress < 50) g_data.mood = ANNOYED;
  else                  g_data.mood = ANGRY;
  DeleteObject(g_data.hbmGorilla);
  g_data.hbmGorilla = RenderGorilla(&g_data);
  LeaveCriticalSection(&g_cs);
}
```

There are some concerns here. First of all, there's the lock hierarchy issue: If reticulating a spline takes the geometry lock, that may violate our lock hierarchy.

If the lock `g_cs` is a hot lock, you may be concerned that all this gorilla stuff will hold the lock for too long. Maybe rendering a gorilla is a slow and complicated operation because it's hard to get the fur just right.

These issues become less onerous if you switch to a lock-free algorithm, but that's an awful lot of work, and it's hard to get right. But maybe you can do just 20% of the work to get 80% of the benefit.

```
void PokeGorilla(double intensity)
{
  // Capture
  EnterCriticalSection(&g_cs);
  GORILLADATA data = g_data; // typo fixed
  LeaveCriticalSection(&g_cs);
  // Recalculate based on captured data
  DeformGorilla(intensity, &data);
  Reticulate(&data.spline);
  int stress = CalculateTension(&data.spline);
  if (stress < 25)      data.mood = RELAXED;
  else if (stress < 50) data.mood = ANNOYED;
  else                  data.mood = ANGRY;
  data.hbmGorilla = RenderGorilla(&data);
  // Commit
  EnterCriticalSection(&g_cs);
  HBITMAP hbmToDelete = g_data.hbmGorilla;
  g_data = data;
  LeaveCriticalSection(&g_cs);
  DeleteObject(hbmToDelete);
}
```

Here, we use the capture/try/commit model. We capture the state of the gorilla into a local variable, then perform our update based on that captured state. The spline reticulation takes place without any locks held, which avoids introducing a lock hierarchy violation. And rendering the gorilla is done without any locks held, which avoids introducing a choke point on the lock. After the calculations are done, we then re-enter the lock and commit the changes.

This pattern uses a last-writer-wins model. If another thread pokes the gorilla while we are still calculating the previous gorilla state, we will overwrite that gorilla state when we complete. For some scenarios, that's acceptable. But maybe the gorilla's emotional state needs to be an accumulation of all the times he's ben poked. We want to detect that somebody has poked the gorilla while we were busy calculating so that we can incorporate that new information into the final result.

To do that, we introduce a change counter.

```
LONG g_lCounter;
void PokeGorilla(double intensity)
{
  BOOL fSuccess;
  do {
    // Capture
    EnterCriticalSection(&g_cs);
    GORILLADATA data = g_data; // typo fixed
    LONG lCounter = g_lCounter;
    LeaveCriticalSection(&g_cs);
    // Recalculate based on captured data
    DeformGorilla(intensity, &data);
    Reticulate(&data.spline);
    int stress = CalculateTension(&data.spline);
    if (stress < 25)      data.mood = RELAXED;
    else if (stress < 50) data.mood = ANNOYED;
    else                  data.mood = ANGRY;
    data.hbmGorilla = RenderGorilla(&data);
    // Commit
    EnterCriticalSection(&g_cs);
    HBITMAP hbmToDelete;
    if (lCounter == g_lCounter)
    {
      hbmToDelete = g_data.hbmGorilla;
      g_data = data;
      g_lCounter++;
      fSuccess = TRUE;
    } else {
      hbmToDelete = data.hbmGorilla;
      fSuccess = FALSE;
    }
    LeaveCriticalSection(&g_cs);
    DeleteObject(hbmToDelete);
  } while (!fSuccess);
}
```

In addition to the regular gorilla data, we also associate a change counter that is incremented each time somebody pokes the gorilla. In real life, you might want to make the change counter part of the `GORILLADATA` structure. (Actually, in real life, you probably shouldn't poke a gorilla.) In a lock-free algorithm, we would `InterlockedCompareExchangeRelease` the lock counter to see if the lock counter changed (and if not, to update it with the new lock counter). But since a `GORILLADATA` structure cannot be updated atomically, we have to use our critical section to perform the comparison-and-update.

Even though we used a lock, we still follow the lock-free pattern. If the gorilla has been poked while we were busy processing our own poke, then we throw away the results of our computations and start over, so that our poke can be accumulated with the previous pokes.

Exercise: What constraints must be applied to the `GORILLADATA` structure for this technique to work?

Raymond Chen

Follow