Why does my program run faster if I click and hold the caption bar?

devblogs.microsoft.com/oldnewthing/20060220-00

February 20, 2006

Raymond Chen

Sometimes, people discover that a long-running task runs faster if you hold down the mouse. How can that be?

This strange state of affairs typically results when a program is spending too much time updating its progress status and not enough time actually doing work. (In other words, the programmer messed up badly.) When you click and hold the mouse over the caption bar, the window manager waits for the next mouse message so it can determine whether you are clicking on the caption or attempting to drag. During this waiting, window painting is momentarily suppressed.

That's why the program runs faster: No window painting means less CPU spent updating something faster than you can read it anyway. Let's illustrate this with a sample program. Start with the new scratch program and make the following changes:

```
class RootWindow : public Window
{
public:
 virtual LPCTSTR ClassName() { return TEXT("Scratch"); }
 static RootWindow *Create();
 void PaintContent(PAINTSTRUCT *pps);
protected:
 LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam);
 LRESULT OnCreate();
 static DWORD CALLBACK ThreadProc(void *p);
private:
 HWND m_hwndChild;
 int m_value;
};
LRESULT RootWindow::OnCreate()
{
 QueueUserWorkItem(ThreadProc, this, WT_EXECUTELONGFUNCTION);
 return 0;
}
void RootWindow::PaintContent(PAINTSTRUCT *pps)
{
 TCHAR sz[256];
 int cch = wnsprintf(sz, 256, TEXT("%d"), m_value);
 ExtTextOut(pps->hdc, 0, 0, 0, &pps->rcPaint, sz, cch, 0);
}
DWORD RootWindow::ThreadProc(void *p)
{
 RootWindow*self = reinterpret_cast<RootWindow*>(p);
 for (int i = 0; i < 100000; i++) {
  self->m_value++;
  InvalidateRect(self->m_hwnd, NULL, NULL);
 }
 MessageBeep(-1);
 return 0;
}
```

This program fires up a background thread that counts up to 100,000 and invalidates the foreground window each time the value changes. Run it and watch how fast the numbers count up to 100,000. (I added a little beep when the loop is finished so you can judge the time by listening.)

Now run it again, but this time, click and hold the mouse on the title bar. Notice that the program beeps almost immediately: It ran faster when you held the mouse down. That's because all the painting was suppressed by the maybe-a-drag-operation-is-in-progress that was triggered when you clicked and held the caption.

Updating the screen at every increment is clearly pointless because you're incrementing far faster than the screen can refresh, not to mention far faster than the human eye can read it. As a rule of thumb, changing progress status faster than ten times per second is generally pointless. The effort you're spending on the screen updates is wasted.

Let's fix our sample program to update at most ten times per second. We will run a timer at 100ms which checks if anything has changed and repaints if so.

```
class RootWindow : public Window
{
 ...
 LONG m_fChanged;
};
DWORD RootWindow::ThreadProc(void *p)
{
 RootWindow*self = reinterpret_cast<RootWindow*>(p);
 for (int i = 0; i < 100000; i++) {
  self->m_value++;
  InterlockedCompareExchangeRelease(&m_fChanged, TRUE, FALSE);
 }
 MessageBeep(-1);
 return 0;
}
LRESULT RootWindow::OnCreate()
{
 QueueUserWorkItem(ThreadProc, this, WT_EXECUTELONGFUNCTION);
 SetTimer(m_hwnd, 1, 100, NULL);
 return 0;
}
LRESULT RootWindow::HandleMessage(
                          UINT uMsg, WPARAM wParam, LPARAM lParam)
{
  ...
  case WM_TIMER:
   switch (wParam) {
   case 1:
    if (InterlockedCompareExchangeAcquire(&m_fChanged,
                                                    FALSE, TRUE)) {
      if (m_value >= 100000) {
        KillTimer(m_hwnd, 1);
      }
      InvalidateRect(m_hwnd, NULL, FALSE);
    }
   }
   break;
   ...
}
```

Instead of updating the screen each time the counter changes value, we merely set a "hey, something changed" flag and check it on our timer. We set the flag with release semantics in the producer thread (because we want all pending stores to complete before the exchange occurs) and clear the flag with acquire semantics in the consumer thread (because we don't want any future stores to be speculated ahead of the exchange).

Run the program again and notice that it counts all the way up to 100,000 instantly. Of course, that doesn't really demonstrate the progress counter, so insert a `Sleep(1);` into the loop:

```
DWORD RootWindow::ThreadProc(void *p)
{
 RootWindow*self = reinterpret_cast<RootWindow*>(p);
 for (int i = 0; i < 100000; i++) {
  self->m_value++;
  InterlockedCompareExchangeRelease(&m_fChanged, TRUE, FALSE);
  Sleep(1);
 }
 MessageBeep(-1);
 return 0;
}
```

This slows down the loop enough that you can now see the values being incremented. It's not the dizzying incrementing that you saw in the original version, but it's fast enough that people will get the point.

The mechanism I used to pass information between the background and foreground thread assumed that background changes were comparatively frequent, so that the timer will nearly always see something worth doing. If you have a mix of fast and slow tasks, you could change the communication mechanism so that the timer shut itself off when it noticed that some time has elapsed with no changes. The background thread would then have to start the timer again when it resumed updating the value. I didn't bother writing this more complicated version because it would just be a distraction from the point of the article.

Raymond Chen

Follow