Closing holes in the update notification pattern

devblogs.microsoft.com/oldnewthing/20130313-00

March 13, 2013

Raymond Chen

Suppose you have a function that is registered to be called the next time something gets updated, and suppose that the notification is a one-shot notification and needs to be re-armed each time you want to wait for the next notification. (For example, the `RegNotifyChangeKeyValue` function behaves this way.) Consider the following code fragment:

```
void onUpdateThing()
{
 // get the updated properties of the thing
 getThingProperties();


 // ask to be called back the next time it updates
 registerUpdateCallback(onUpdateThing);
}


mainProgram()
{
 // get the thing's initial properties
 // and register for updates
 onUpdateThing();
}
```

There is a race condition here if the thing updates twice in rapid succession. On the first update, your `onUpdateThing` function is called. If the second update occurs *while* `getThingProperties` *is running*, then your call to `registerUpdateCallback` will be too late, and you will miss the second update.

The solution is to register for the next update *before* studying the previous one.

```
void onUpdateThing()
{
 // ask to be called back the next time it updates
 registerUpdateCallback(onUpdateThing);


 // get the updated properties of the thing
 getThingProperties();
}
```

That way, if a second update comes in while you're studying the first one, your update callback will be called because you already registered it. (I'm assuming you're only interested in the last update.)

Of course, this assumes that update requests are queued if the receiving thread is busy. If updates can be received during the execution of `getThingProperties`, then you will end up in a bad re-entrant situation: During the processing of one update, you start processing a new update. Then when the nested update finishes, you return to the original update, which is now actually performing the second half of the second update.

Suppose your update code wants to keep the colors of two additional objects in sync with the color of the thing:

```
void getThingProperties()
{
 Color currentThingColor = getThingColor();
 object1.setColor(currentThingColor);
 object2.setColor(currentThingColor);
}
```

If the `setColor` method creates a re-entrancy window, you can have this problem:

- Thing changes color to red.
- `onUpdateThing` begins.
- Register update callback.
- `getThingProperties` reads current color as red.

- `getThingProperties` sets object 1's color to red. The `setColor` method creates an opportunity for re-entrancy by some means. (For example, it may send a message to another thread, causing inbound sent messages to be processed.)
    - Thing changes color to blue.
    - `onUpdateThing` begins.
    - Register update callback.
    - `getThingProperties` reads current color as blue.
    - `getThingProperties` sets object 1's color to blue.
    - `getThingProperties` sets object 2's color to blue.
    - `getThingProperties` returns.
    - `onUpdateThing` returns.
- `getThingProperties` sets object 2's color to *red*. (Oops.)
- `getThingProperties` returns.
- `onUpdateThing` returns.

One solution is to use a sequence number (also known as a change counter) that gets incremented each time the thing changes. If there is only one thread which updates the thing, you can try to update it atomically. For example, if the information is in the registry, you can put all the information into a single registry value or use registry transactions.

If you can associate a change counter with the data, then you can use the following algorithm:

```
// start with a known invalid value
// (If you have multiple listeners, then this naturally
// needs to be instance data rather than global.)
LONG lLastChange = 0;


void onUpdateThing()
{
 bool finished = false;
 do {
  // record the most recent change we've processed
  lLastChange = getThingChangeCount();


  getThingProperties();


  // ask to be called back the next time it updates
  registerUpdateCallback(onUpdateThing);


  // did it change while we were busy?
  LONG lNewChange = getThingChangeCount();


  finished = lLastChange == lNewChange;
  if (!finished) {
   // cancel the update callback because we don't
   // want to be re-entered
   unregisterUpdateCallback(onUpdateThing);
  }
 } while (!finished);
}
```

Another solution would be to detect the re-entrancy and just remember that there is more work to be done after the previous update finishes.

```
// 0 = not busy
// 1 = busy
// 2 = busy, and a change occurred while we were busy
// (If you have multiple listeners, then this naturally
// needs to be instance data rather than global.)
int iBusy = 0;


void onUpdateThing()
{
 // ask to be called back the next time it updates
 registerUpdateCallback(onUpdateThing);


 if (iBusy) {
   iBusy = 2;
 } else {
  iBusy = 1;
  do {
   getThingProperties();
  } while (–iBusy);
 }
}
```

Note that all of the above examples assume that the `onUpdateThing` function has thread affinity.

Raymond Chen

Follow