Subject: WhenAction() -> ProcessEvents() -> WhenAction() -> hang, crash :-)
Posted by xrysf03 on Sun, 08 Dec 2019 14:21:49 GMT
View Forum Message <> Reply to Message

Dear everybody,

while playing with the GUI in Ultimate++ (still without forking my own threads), lately it feels like I'm stuck... like I've found myself in a place where I'm stretching things a little too far, and I need to establish a new direction.

To be more specific:

I'll attach just a screenshot. Mind the large "Go" button in the upper right corner, and the large ScatterCtrl. After setting up some parameters, the user needs to press "Go" and the key things start to happen: the proggie tunes into a sequence of frequencies, spaced by 2 MHz in this setup, calculates an FFT for each frequency and applies the data to the final plot (ScatterCtrl).

In other words, the callback handler, installed for the "Go" button's WhenAction, does a lot of time-consuming work, blocks the GUI for quite some time. To allow for a gradual display of the data, I've learned to call Ctrl::ProcessEvents() at suitable instants during the band scan: namely, just after one partial FFT gets applied to the visual output (the spectrum visually grows from left to right). So the band scanning callback is calling Ctrl::ProcessEvents() every now and then.

Here's the catch: when the user presses the "Go" button again, while the band scanning routine is still at work (= the previous GUI callback hasn't returned yet), the program stops working - hangs and either has to be killed or segfaults on its own in a few seconds.

I figured right away that I probably should not call the button callback again, even if this is just a recursive reentrancy, rather than multi-threaded reentrancy. I tried to avoid the problem in two ways:

1) I tried installing a "latch" at the very beginning of the heavy "band scan" function, with a boolean flag (member of my main window class) to signal if the band scan is already running - in which case the second (and further) invocations of that heavy function would return immediately, without doing anything. That hasn't helped - and I could hazard a guess that this has to do with compiler options such as a missing -D_REENTRANT for GCC. This possibly controls how local variables are allocated and initialized (= not necessarily on stack etc).

2) I tried re-installing a different callback on start of the heavy worker function (itself the initial "Go" button callback). I just used the THISBACK macro at the start of this fat callback, to install *another* dummy callback. (I'd also change the label on the button to "Stop", and tried raising another flag, that would signal to the fat band scan function to give up the sweep after it's finished processing the current frequency slot.) For some reason, installing a different callback didn't work either, the result is the same = a hang upon the second "Go" button click.

Mind the checkbox/Option labeled "loop around". That one works fine. It doesn't have its WhenAction() hooked: the code of my fat band scan function just polls its value every now and then, and quits looping if the check mark is removed. So: that works fine, which makes me believe, that it's the auto-recursive calling of the WhenAction of the "Go" button that causes the hang.

If I don't violate the rule that "I should not click the Go button while already scanning", the program works fine and is actually getting useful already.

I don't believe very much that these "hangs when clicking Go too early again" would be a problem inside my own code. I'm inclined to believe that this behavior is a "feature" = I'm trying to twist the GUI clockwork a little too far.

And, I should probably "do the right thing" = instead of abusing ProcessEvents() excessively, I should probably just fork my own thread for the heavy work, and return control to the GUI in the foreground thread ASAP. The GUI callbacks should not block. That way, the GUI will remain alive and snappy. I recall reading an UPP forum post about some way to "ping back" the GUI front end thread to run some code in its context, and that there's a GUI lock that I need to take if I mess with the objects from a different thread directly (actually maybe that's not possible at all).

= I probably know roughly what I need to do.
And the reason for me to post this topic is that: any comments are welcome :)

Frank

File Attachments

1) rtlsdr_skyline_GUI.png, downloaded 334 times

Subject: Re: WhenAction() -> ProcessEvents() -> WhenAction() -> hang, crash :-)
Posted by koldo on Sun, 08 Dec 2019 15:22:44 GMT
View Forum Message <> Reply to Message

Dear Frank

I would try not to use ProcessEvents but for projects to be used "at home" or in beta.
If you have slow jobs, it is may be better to use threads.
This way the "Go" button would be just a starter, and after that it would be good to disable it, or to change its behaviour to a "Stop".
You can imagine you are doing like a sound player, with play and stop keys.

Best regards
Iñaki

Subject: Re: WhenAction() -> ProcessEvents() -> WhenAction() -> hang, crash :-)
Posted by xrysf03 on Sun, 08 Dec 2019 19:57:29 GMT
View Forum Message <> Reply to Message

Thanks for resonse Koldo, always polite and always swift :)

Hmm... I can see that U++ has its own class Mutex and ConditionVariable. And they appear to be less convoluted (objectified) than the C++11 std::mutex and std::condition_variable (geez... if I didn't know the bare C libpthread version, I would probably just shake my head in disbelief).

My favourite and perhaps "naive" mechanism for passing lumps of something to do, between a producer and a consumer, is using a queue, protected by a mutex+condvar. What is the most appropriate container in U++ to implement a FIFO queue? The online help mentions Vector, Array, BiVector, BiArray... and in this case, I don't need to index the "collection" on anything, all I need is FIFO operation. So I don't need a Set, let alone a Map.
Hmm. Or perhaps I'll just preallocate the buffers and use a simple Semaphore...

Actually none of this is probably relevant to the GUI. I'm just thinking forward :) I'll probably use one background thread to do the tuning and grabbing, and another background thread to do the crunching and ScatterCtrl updates. And I'll write the updates straight into the buffer that's been preallocated and passed to the ScatterCtrl. So all I need to ask the GUI foreground thread to do, is use Ctrl::Call() or PostCallback() to do a ScatterCtrl::Refresh() on my spectrogram object... any examples of this would be welcome. Or generally how to tell the GUI foreground thread to redraw a ScatterCtrl that has had its data buffer changed by a background thread...

Ahh, apologies, I'll have to check the reference/GuiMT . There's a nice example of PostCallback(). Ouch... anonymous functions in C++...

Subject: Re: WhenAction() -> ProcessEvents() -> WhenAction() -> hang, crash :-)
Posted by mirek on Mon, 09 Dec 2019 07:55:57 GMT
View Forum Message <> Reply to Message

- latch should work. I am doing that approach all the time. "REENTRANT" is irrelevant, reentrant code always works (there is a lot of reentrancy everywhere in U++)
- from user perspective, you should also disable the button and reenable at the end of task, but I recommend to keep latch (I call them "lock") anyway
- I usually make lock as counter
- From what you wrote (maybe I have missread), you are not running that task as background thread. In that case, Mutex is not likely to help you.

In short: latch/lock should work, if it does not, look for problem elsewhere.

Posting testcase would help :)

Subject: Re: WhenAction() -> ProcessEvents() -> WhenAction() -> hang, crash :-)
Posted by Oblivion on Mon, 09 Dec 2019 11:00:59 GMT
View Forum Message <> Reply to Message

Hello Frank,

Adding to what Koldo and Mirek said,

Quote:
What is the most appropriate container in U++ to implement a FIFO queue? T

IME, BiVector for FIFO/LIFO queue of moveable types, and BiArray for FIFO/LIFO queue of non-moveable/quirky types.
BiVector and BiArray are bi-directional random access containers, allowing adding or removing elements from both ends in a constant amortized time.

Quote:
Hmm... I can see that U++ has its own class Mutex and ConditionVariable. And they appear to be less convoluted (objectified) than the C++11 std::mutex and std::condition_variable (geez... if I didn't know the bare C libpthread version, I would probably just shake my head in disbelief).

IMHO, as I said elsewhere, you should really look into Upp::AsyncWork or Job. I'd suggest using Job here, not because I am the author of it :lol: , but from what you describe, it seems that it suits your needs better:

- Unlike Upp::AsyncWork, it does not require a pre-allocated thread-pool. It is a single, scope-bound worker thread that can be used as a dedicated thread.

- Unlike Upp::AsyncWork, it is guaranteed to run the desired function in another thread.

- Unlike Upp::Asyncwork, it has an internal latch: if a work is already in progress, then calls to Job::Do method will simply return false.

- Just like AsyncWork it has a result gathering, thread cancellation and exception propagation mechanisms.

I've attached the Job package and the multithreaded version of U++'s AnimatedHello example, running on Job. It demonstrates a one way of constantly updating the display from another thread, and handling gui locks/threads. Hope it helps.

(Note it can also be written almost identically using Upp::AsyncWork. But you'll have to provide a latch yourself, and also it will bring in the overhead of a pre-allocated thread pool)

Use CTRL + D to start animation, and CTRL + C to stop animation.

And let me know if you encounter any problem or have any questions.

Best regards,
Oblivion

File Attachments

1) Example.zip, downloaded 288 times