Subject: Thread calls GUI
Posted by Sami on Sat, 14 Feb 2009 18:26:01 GMT
View Forum Message <> Reply to Message

I realize a thread cannot call GUI in upp. It is not however clear how threading should be implemented then. I would ask help for proper solution to the example given below.

```
struct Interface {
  virtual int Ask ( const char * ) = 0;
};

struct Work {
  Interface *gui;
};

struct Library {
  Library ( Work w ) {
    int a = w.gui->Ask ( "Ok?" );
  }
};

void Threading ( Work w ) {
  Library ( w );
}

struct Task
:MyTask<TopWindow>
,Interface {
  typedef Task CLASSNAME;
  Task() {
    CtrlLayout(*this, "Example" );
    Work w;
    w.gui = this;
    Thread().Run ( callback1 ( Threading, w ) );
  }
  volatile Atomic q;
  int Ask_Weird_Hacked ( const char *s, unsigned dummy ) {
    return q = 1 + PromptYesNo ( String().Cat() << s );
  }
  int Ask ( const char *s ) {
    //problem here, cannot call PromptYesNo()
    q = 0;
    PostCallback ( callback2 ( this, &Task::Ask_Weird_Hacked, s, 0 ) );
    while ( !q ) Sleep ( 10 );
    return q - 1;
  }
};
```

So we begin with Task() and our problem is how to implement Ask() call properly. I first understood the Gate-method is what I'm looking for, but I didn't get it to work, can somebody explain what is it? The manual was in my opinion incomplete here.

Subject: Re: Thread calls GUI
Posted by mirek on Sun, 15 Feb 2009 07:24:18 GMT
View Forum Message <> Reply to Message

Unfortunately, Gate cannot work here. Gate is supposed to return the value, which is not possible until callback is performed..

I am afraid that the solution for your problem might be quite complex.

IMO, you will have to use something like Semaphore on thread part. Use PostCallback to signal to GUI thread you need that prompt, enter semaphore after PostCallback.

GUI thread then performs the prompt, signals the result via some shared variable, then releases semaphore of thread to get it going.

```
void DoAsk(Semaphore *sem, int *result)
{
   *result = PromptYesNo("");
   sem->Release();
}

struct MyThread {
    int Ask() {
        Sempahore sem;
        int result;
        PostCallback(callback2(DoAsk, &sem, &result));
        sem.Wait();
        return result;
    }
}
```

(To my best knowledge, we do not need mutex for result, as sempahore does the synchronization for us as well).

(Not testes, but should work).

Mirek

Subject: Re: Thread calls GUI
Posted by Sami on Sun, 15 Feb 2009 21:55:22 GMT
View Forum Message <> Reply to Message

Thanks for your answer. It appears that this semaphore and additional function should be done for all interface calls... Have you considered fixing this apparent design issue in upp? Why we cannot have upp serialize (if it needs to) the gui calls transparently, so that there would be no limits for threads calling the gui?

Additional question. Suppose we have a kill button in the GUI. void Task::KillButton(). How to kill the thread in this function? We presume the thread is heavy and cannot ping asking the interface for ShouldWeCancelNow() frequently enough to be able to shutdown itself.

Subject: Re: Thread calls GUI
Posted by Mindtraveller on Sun, 15 Feb 2009 22:22:57 GMT
View Forum Message <> Reply to Message

Some time ago I`ve started developing "alternative" multithreading system for U++. General idea is that you do not need any sync objects. Because threads do not see ANY shared variables. Instead threads have internal callback queues and exchange with callbacks. Realization is rather optimal (but could be better if I had more spare time).

Recently I was badly needed this approach to be working for a number of threads and also for a "main" GUI thread. Finally these classes are ready and tested for some time, but still under heavy development.

Simplified code looks like this...

1. Declaring main GUI thread/window and one more thread:

```
class GUIThread : public CallbackQueue, public WithMainWindowLayout<TopWindow>
{
public:
 GUIThread();
   ~GUIThread();
 virtual void Init();
 virtual void Shutdown();

 void HandleIncomingMessages();

public /*sync*/:
 void RefreshAll(const Drawing &bdr, const ControlGUI &cg);
 void SetupBathsSettings(Vector<Vector<Value> > rows);
};

class IOThread : public CallbackThread, protected RS232
{
```

```
public:
 IOThread();
   ~IOThread();
 virtual void Init();
 virtual void Shutdown();

public /*sync*/:
 void GetAOpState(byte addr);
};

Main app function is simple:
GUIThread     guiThread;
IOThread      ioThread;
ControlThread controlThread;

GUI_APP_MAIN
{
 try
 {
  CallbackQueue::InitAll();
  CallbackQueue::StartAll();

  guiThread.Sizeable().Run();

  CallbackQueue::ShutdownAll();
 }
 catch (const Exc &ex)
 {
  PromptOK(ex);
 }
}


Finally, if I want any of my threads (including main/GUI) to do something, I just request for this:
// i/o therads checks AOp devices and tells their availability to Control thread
void IOThread::GetAOpState(byte addr)
{
 for (int i=0; i<attempts; ++i)
 {
  protoSend[3] = addr;
  protoSend[4] = CMD_AOP_STATUS;
  protoSend.Send(*this, timeout);

  if (protoRecv.Receive(*this, timeout))
  {
   if ((int)protoRecv[3] != (int)addr)
    continue;
   controlThread.Request(&ControlThread::AOpStatus, addr, protoRecv[4]);
```

```
  }
 }

 controlThread.Request(&ControlThread::AOpUnavailable, addr);
}

// Control thread analyzes system state and updates GUI accordingly
void ControlThread::SetupDisplayDrawing(bool enableSettings)
{
 static ControlGUI cg;
 // setting controls to be enabled/disabled
 // ...


 // drawing system elements and parameters
 DrawingDraw d(DISPLAY_W,DISPLAY_H);
 // ...

 guiThread.Request(&GUIThread::RefreshAll, static_cast<Drawing>(d), cg);
}
```

...and no sync objects with their debug.
If this is handy for you, I`ll upload these "alternative" multithreading sources here.

Subject: Re: Thread calls GUI
Posted by mirek on Sun, 15 Feb 2009 22:27:43 GMT
View Forum Message <> Reply to Message

Sami wrote on Sun, 15 February 2009 16:55Thanks for your answer. It appears that this semaphore and additional function should be done for all interface calls... Have you considered fixing this apparent design issue in upp? Why we cannot have upp serialize (if it needs to) the gui calls transparently, so that there would be no limits for threads calling the gui?

Yes, it is planned.

Quote:
Additional question. Suppose we have a kill button in the GUI. void Task::KillButton(). How to kill the thread in this function? We presume the thread is heavy and cannot ping asking the interface for ShouldWeCancelNow() frequently enough to be able to shutdown itself.

I am afraid that killing the thread is not as easy as it might seem - who would free all associated resources?! (e.g. allocated memory, opened files and sockets...).

That is why I think that some form of "ShouldWeCancelNow" is in fact necessarry. It can take the

form of periodic call to some function which in case of cancel throws exception - that IMO is the most effective way.

Mirek

Subject: Re: Thread calls GUI
Posted by Sami on Mon, 16 Feb 2009 20:35:46 GMT
View Forum Message <> Reply to Message

Thanks for the replies.

Ok, I cannot kill the thread. What about thread exceptions. How should they be implemented. I have an option to replace c++ exceptions with a call to static function if needed. Are the exceptions allowed and where should I catch them (in the example I gave at the top post)? It's not trivial to get rid of the exceptions in the thread (to exit cleanly).

Subject: Re: Thread calls GUI
Posted by mirek on Tue, 17 Feb 2009 06:36:16 GMT
View Forum Message <> Reply to Message

Sami wrote on Mon, 16 February 2009 15:35Thanks for the replies.

Ok, I cannot kill the thread. What about thread exceptions. How should they be implemented. I have an option to replace c++ exceptions with a call to static function if needed. Are the exceptions allowed and where should I catch them (in the example I gave at the top post)? It's not trivial to get rid of the exceptions in the thread (to exit cleanly).

But exceptions are GOOD in this context. They would perform the necessarry cleanup of resources. Simply catch the "thread canceled" exception in the main thread routine...

In fact, the only hard part is to how to throw them. There I see no other option than to call some function periodically, check for the exception flag and throw if set.

Well, anyway, I guess any serious GUI program should show the progress of processing anyway - maybe that is the right place to check for cancelation too....

Mirek

Subject: Re: Thread calls GUI

Posted by tojocky on Tue, 17 Feb 2009 14:57:30 GMT
View Forum Message <> Reply to Message

How about to add semaphore method dword Semaphore::Wait(int timeout)?

in win32 is simple:

```
int Semaphore::Wait( int timeout )
{
 dword result_value;
 result_value = WaitForSingleObject(handle, timeout);
 if(result_value == WAIT_FAILED) return(SEMAPHORE_WAIT_ERROR);
 if(result_value == WAIT_TIMEOUT) return(SEMAPHORE_TIMEOUT);
}
```

but in POSIX is more hardly.
The good article found here.

I think it have sense on i have a postcallback and i know maximum execution time. Or I'm wrong?

Subject: Re: Thread calls GUI
Posted by mirek on Tue, 17 Feb 2009 18:18:28 GMT
View Forum Message <> Reply to Message

tojocky wrote on Tue, 17 February 2009 09:57How about to add semaphore method dword Semaphore::Wait(int timeout)?

in win32 is simple:

```
int Semaphore::Wait( int timeout )
{
 dword result_value;
 result_value = WaitForSingleObject(handle, timeout);
 if(result_value == WAIT_FAILED) return(SEMAPHORE_WAIT_ERROR);
 if(result_value == WAIT_TIMEOUT) return(SEMAPHORE_TIMEOUT);
}
```

but in POSIX is more hardly.
The good article found here.

I think it have sense on i have a postcallback and i know maximum execution time. Or I'm wrong?

I do not know. I believe all these timeouts just make it more error-prone. You generally should not depend on timeout when dealing with semaphore (IMO!).