Subject: Re: Question about PostCallback from Child Thread
Posted by kfeng on Fri, 13 Jul 2007 15:19:09 GMT
View Forum Message <> Reply to Message

luzr wrote on Thu, 12 July 2007 21:54kfeng wrote on Thu, 12 July 2007 00:11OK, suppose I have a C struct with a bunch of pointers to the heap:

```
struct
{
  int    *intP;
  double *doubleP;
  char   **strP;
...
}
```

It's filled out by the child process and I pass a pointer to an instance to PostCallback(). Will PostCallback() be smart enough to lock all the pointed-to members? If not, is there a way to make the child thread block and wait until the parent is done reading?

The problem is **strP - I need this to run fast so I don't want to be looping through the members locking each one by hand - may be simpler to just get the child to wait for the parent to finish. Is there a way I can do this to a child thread?

Now I am a little bit confused.... First of all, "pointers to the heap" sounds like a bad practice if you are following U++ path of things.... Anyway, surely can happen.

Then the question is what "locking each one by hand" is supposed to mean.

In MT, what you have to lock (using mutex) is data that at specific moment can be accessible by more than single thread. Is this the case? Note that if you e.g. create heap data than are only passed using PostCallback and are not references anywhere else, you do not need to lock.

Now another question is why do you want to lock one by one? You can have single mutex protecting the whole array.... and lock just before the loop, unlock after.

Mirek

OK. My apologies. Let me start from the beginning. I am using a 3rd party C library for retrieving real-time financial tickdata. In my first non-GUI pure-C non-MT program. the sequence looks like this:

1. Connect to server
2. Blocking wait for the server to send tickdata
3. Read the tickdata
4. Release the memory using their library function (since they did the allocation, they need me to release when I'm done)

5. Goto 2

Now the ugly part is the data. It doesn't come in one big block. Instead it looks like this:

```
typedef struct _x {
  int nCols;
  int nRows;
  union {
    double* pD;
    int*    pI;
    char**  pS;
    ...
  }
} *xArray;
```

So you can imagine that xArray is an array of columns represented by pointers to different types. In other words, xArray represents a "table" as an array of individually contiguous chunks of memory. xArray[0].pI[0] represents the first integer is the first column, if the first column is known to be an integer type. xArray[0].pI[3] represents the fourth integer in the first column. Hence, we know that xArray[0].pI is one contiguous chunk of memory. It gets ugly if we have pS, I think. xArray[1].pS[1] is the second string in the second column which is a contiguous list of strings. As I understand it, locking xArray[0].pI is sufficient, but I must lock each of xArray[1].pS[0..(nRows-1)].

But maybe this is where my thinking is wrong. My confusion lies in the wrong assumption that I needed to lock all the little malloc'ed pieces of memory. After rethinking the problem, and rewriting this e-mail over many interations, I can understand your confusion (because >I< was confused!).

A. xArray result = ReadFromLibrary_Blocking(); // Child Thread
B. call PostCallback(callback(..., result); // Asynchronous in Child Thread
C. Goto A. // Child Thread

X. Read result into GUI // Parent GUI Thread
Y. Free result // Parent GUI Thread
Z. Automatically unlocks result on exit from method // Parent

Here are the facts:
* Child Thread creates new result for every A.
* Parent won't read result until GUI is ready and result is in the PostCallback() queue.
* Parent will free result when it's done and Child never cares because once it's passed into the parent, it never reads/writes to the variable again.

So what is pointed to by result will never be read/written by two different threads. Information flows in one direction, sequentially from child thread(s) to parent GUI thread, so in fact, it's >NOT< the "locking" service that is required from PostCallback - rather, it's the queueing mechanism that's important in this case. In the worst-case-scenario, if the child process runs super-fast

relative to the GUI, the "result" variable in the child gets overwritten many times, but the parent doesn't care - "result"'s value (ie the address to malloc'ed memory) is simply queued up by PostCallback() for later consumption.

Thank you for taking the time out of your busy schedule to explain and to read this. I really learned a great deal!

Regards,
Ken